

Aztec CG65
Cross Development Software
for 65xx-based Systems

version 3.2

July 1987

Copyright (c) 1986 by Manx Software Systems, Inc.

All Rights Reserved

Worldwide

Distributed by:
Manx Software Systems, Inc.
P.O. Box 55
Shrewsbury, N.J. 07701
201-542-2121

USE RESTRICTIONS

The components of the Aztec CG65 software development system are licensed software products. Manx Software Systems reserves all distribution rights to these products. Use of these products is prohibited without a valid license agreement. The license agreement is provided with each package. Before using any of these products the license agreement must be signed and mailed to:

Manx Software Systems
P. O. Box 55
Shrewsbury, N. J 07701

The license agreement limits use of these products to one machine. Any uses of these products that might lead to the creation of or distribution of unauthorized copies of these products will be a breach of the licensing agreement and Manx Software Systems will exercise its right to reclaim the original and any and all copies derived in whole or in part from first or later generations and to pursue any appropriate legal actions.

Software that is developed with Aztec CG65 software development system can be run on machines that are not licensed for these products as long as no part of the Aztec C software, libraries, supporting files, or documentation is distributed with or required by the software. In the latter case a licensed copy of the appropriate Aztec C software is required for each machine utilizing the software. There is no licensing required for executable modules that include runtime library routines.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. DAC #84-1, 1 March 1984. DOD Far Supplement.

COPYRIGHT

Copyright (C) 1981, 1982, 1984, 1986 by Manx Software Systems. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without prior written permission of Manx Software Systems, Box 55, Shrewsbury, N. J. 07701.

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Manx Software Systems to notify any person of such revision or changes.

TRADEMARKS

Aztec CG65, Manx AS65, and Manx LN65, are trademarks of Manx Software Systems. CP/M-86 is a trademark of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of Bell Laboratories. Macintosh is a trademark of Apple Computer.

Manual Revision History

Aug 1984	First Edition
Jan 1986	Second Edition
Aug 1986	Third Edition
July 1987	Fourth Edition

Summary of Contents

65xx-specific Chapters

<i>title</i>	<i>code</i>
Overview	ov
Tutorial Introduction	tut
The Compilers	cc
The Assemblers	as
The Linker	ln
Utility Programs	util
Library Generation	libgen
Technical Information	tech

System Independent Chapters

Overview of Library Functions	libov
System-Independent Functions	lib
Style	style
Compiler Error Messages	err

Index

Index	index
-------------	-------

Contents

Overview	ov
Tutorial Introduction	tutor
1. Installing Aztec CG65	3
2. Creating Object Module Libraries	5
3. Translating a program into Intel hex code	7
4. Special Features	11
4.1 Native code vs. pseudo code	11
4.2 Zero-page usage	12
5. Where to go from Here	13
The compilers	cc
1. Operating Instructions	3
1.1 The C Source File	3
1.2 The Output Files	4
1.3 Searching for <i>#include</i> files	6
2. Compiler Options	7
2.1 Summary of Options	7
2.2 Description of Options	9
3. Programmer Information	14
3.1 Supported Language Features	14
3.2 Structure Assignment	14
3.3 Line Continuation	14
3.4 The <i>void</i> Data Type	14
3.5 Special Symbols	15
3.6 String Merging	15
3.7 Long Names	16
3.8 Reserved Words	16
3.9 Global Variables	16
3.10 Data Formats	17
3.11 Floating Point Exceptions	18
3.11 Register Variables	20
3.12 In-line Assembly Language Code	20
3.13 Writing Machine-Independent Code	21
4. Error Processing	23
The Assemblers	as

1. Operating Instructions	3
1.1 The Source File	3
1.2 The Object Code File	4
1.3 Listing File	4
1.4 Searching for <i>instxt</i> Files	4
2. Assembler Options	5
3. Programmer information	5
The Linker	ln
1. Introduction to linking	3
2. Using the Linker	7
3. Linker Options	9
Utility Programs	util
arcv	4
cnm65	5
crc	9
hd	10
hex65	11
lb65	14
make	25
mkarcv	4
obd65	43
optint65	44
ord65	45
sqz65	46
Library generation	libgen
1. Rewriting the functions	3
1.1 The start-up function	3
1.2 The <u>main</u> function	4
1.3 The Unbuffered i/o functions	4
1.4 The standard i/o functions 'agetc' and 'aputc'	9
1.5 The <i>sbrk</i> heap management function	9
1.6 The <i>exit</i> and <u>exit</u> functions	9
2. Building the libraries	10
3. Function descriptions	11
Technical Information	tech
1. Memory Organization	4
2. Overlays	7
3. Interfacing to Assembly Language	14
4. Object Code Format	18
5. The Pseudo Stack	29
Overview of Library Functions	libov
1. I/O Overview	4
1.1 Pre-opened devices, command line args	4
1.2 File I/O	6

1.2.1 Sequential I/O	6
1.2.2 Random I/O	6
1.2.3 Opening Files	6
1.3 Device I/O	7
1.3.1 Console I/O	7
1.3.2 I/O to Other Devices	7
1.4 Mixing unbuffered and standard I/O calls	7
2. Standard I/O Overview	9
2.1 Opening files and devices	9
2.2 Closing Streams	9
2.3 Sequential I/O	10
2.4 Random I/O	10
2.5 Buffering	10
2.6 Errors	11
2.7 The standard I/O functions	12
3. Unbuffered I/O Overview	14
3.1 File I/O	15
3.2 Device I/O	15
3.2.1 Unbuffered I/O to the Console	15
3.2.2 Unbuffered I/O to Non-Console Devices	16
4. Console I/O Overview	17
4.1 Line-oriented input	17
4.2 Character-oriented input	18
4.3 Using ioctl	19
4.4 The sgtty fields	19
4.5 Examples	20
5. Dynamic Buffer Allocation	22
6. Error Processing Overview	23
System Independent Functions	lib
Index	5
The functions	8
Style	style
1. Introduction	3
2. Structured Programming	7
3. Top-down Programming	8
4. Defensive Programming and Debugging	10
5. Things to watch out for	15
Compiler Error Codes	err
1. Summary	4
2. Explanations	7
3. Fatal Error Messages	35
Index	index

OVERVIEW

Overview

The Aztec CG65 Software Development Package is a set of programs for developing programs in the C programming language; the resulting programs run on ROM- and/or RAM-based systems that use a 65xx microprocessor. The development can be done on several host systems, as defined below.

Some of the features of Aztec CG65 are:

- * The full C language, as defined in the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, is supported.
- * Two pairs of compilers and assemblers are provided. One pair generates native 65xx or 65C02 code, and the other "pseudo code". A program's native code is directly executed by the processor, while its pseudo code is executed by an Aztec routine that is in the program. A program can contain both native and pseudo code.
- * An extensive set of user-callable functions is provided, in source form. To use these functions, you must first compile and assemble them, and create libraries of the resulting object modules. To use the standard and/or unbuffered i/o functions, you'll have to rewrite the unbuffered i/o functions, which are designed for an Apple // ProDOS system.
- * Code can be partitioned into overlays, allowing programs to be created and executed that are larger than available memory. To use this feature, you must rewrite the unbuffered i/o functions.
- * Modular programming is supported, allowing the components of a program to be compiled separately, and then linked together.
- * Assembly language code can either be combined in-line with C source code, or placed in separate modules which are then linked with C modules.
- * Special features are provided for programs that are to be burned into ROM: (1) a utility program is provided that will generate Intel hex records for a program. ROM chips generated from these records will contain the program's code, a copy of its initialized data, and optionally, in the 65xx power-up and interrupt vector fields, pointers to the routine that handle these events; (2) a ROM program can contain both

initialized and uninitialized global and static variables. When the program starts, its initialized variables will be automatically set from the copy in ROM, and its uninitialized variables will be cleared.

In order to create fast-executing programs, the compilers generate code that use variables in the zero page of the 65xx. Since each 65xx-based system uses different sections of the zero page, the compilers allow you to specify the locations in the zero page that will be used by your programs.

The functions provided with this package are UNIX compatible and are compatible with Aztec C packages provided for other systems. Thus, once you have customized the functions, you can create programs that will run on UNIX-based systems or on other systems supported by Aztec C with little or no change.

Host systems

The Aztec CG65 software runs on several host systems, including:

- * PCDOS/MSDOS systems, such as the IBM PC;
- * Vax systems that use the Ultrix operating system;
- * PDP-11 systems that use UNIX version 7 or later

Components

Aztec CG65 contains the following components:

- * *cg65* and *as65*, the native-code compiler and assembler;
- * *cci* and *asi*, the interpretive-code compiler and assembler;
- * *ln65*, the linker;
- * *lb*, the object module librarian;
- * Source for the library functions;
- * Several utility programs.

Preview

This manual is divided into two sections, each of which is in turn divided into chapters. The first section presents 65xx-specific information; the second describes features that are common to all Aztec C packages. Each chapter is identified by a symbol.

The 65xx-specific chapters and their identifying codes are:

tutor describes how to get started with Aztec CG65; it discusses the installation of Aztec CG65, and gives an overview of the process for turning a C source program into Intel hex code;

cc, *as*, and *ln* present detailed information on the compilers, assemblers, and linker;

util describes the utility programs that are provided with Aztec CG65;

libgen describes the creation of object module libraries from the provided source;

tech discusses several miscellaneous topics, including memory organization, overlays, writing assembly language functions, and object module format.

The System-independent chapters and their codes are:

libov presents an overview of the system-independent features of the functions provided with Aztec CG65;

lib describes the system-independent functions provided with Aztec CG65;

style discusses several topics related to the development of C programs;

err lists and describes the error messages which are generated by the compiler and linker.

TUTORIAL INTRODUCTION

Chapter Contents

Tutorial Introduction tutor

1. Installing Aztec CG65 3

2. Creating Object Module Libraries 5

3. Translating a program into Intel hex code 7

4. Special Features 11

 4.1 Native code vs. pseudo code 11

 4.2 Zero-page usage 12

5. Where to go from Here 13

Tutorial Introduction

This chapter describes how to quickly start using your Aztec CG65 cross development software. We first present the steps to install the Aztec CG65 software on your disks. We then briefly mention the fact that you must generate object module libraries from the source that comes with Aztec CG65, and refer you to the chapter in which this is discussed. Then we describe the steps to translate a C program to Intel hex code. Finally, we introduce the rest of the manual.

Ideally, this chapter should consist of a cookbook set of steps that you can follow to get started using Aztec CG65. However, since one of those steps is a long and involved one, (ie, to modify the library functions and then generate libraries), we recommend that you follow the first step, which leads you through the installation of Aztec CG65 on your system, and then simply read the rest of chapter to get a idea of how programs are developed using Aztec CG65. Then you can read the chapter that discusses library generation, make any needed revisions to the library function source, and generate your libraries. Finally, you can come back to this chapter and translate a C program into Intel hex code.

1. Installing Aztec CG65

To install Aztec CG65 on your system, copy the files from the distribution media (disk or tape) onto your disks.

If your system is one (such as the IBM PC running PCDOS, or a UNIX system) that supports a hierarchical directory structure, we recommend that you place the Aztec CG65 software in a set of related directories, as shown in the following diagram.

Directory CG65	Contents
BIN	executable programs
INCLUDE	header files
LIB	object module libraries
STDIO	stdio.arc files
MCH65	mch65.arc files
MISC	misc.arc files
PRODOS	prodos.arc files
DEV	dev.arc files
TIME	time.arc files
OVLY	ovly.arc files
ROM	rom.arc files
UTILITY	
XFER	xfer.arc files
TTY	tty.arc files
CONFIG	config.arc files
WORK	your programs

Copy the Aztec CG65 files into the directories as follows:

- * Into the BIN directory, copy all executable Aztec CG65 programs.
- * Into the INCLUDE directory, copy all "include files" (that is, files having extension *.h*).
- * Into the LIB directory, copy the source archive *libmake.arc*. The libraries that you create will reside in this directory.

Extract the files from this archive using the *arcv* command, and then delete *libmake.arc* from the LIB directory.

To extract files from *libmake.arc* follow these steps: (1) make sure that the BIN directory is in the path of directories that will be searched by the operating system for programs (on PCDOS and UNIX, this means adding the BIN directory name to the PATH environment variable); (2) enter the appropriate command to make LIB the default or current directory (for example, on PCDOS this command is *cd \CG65\LIB*); (3) enter the command *arcv libmake.arc*.

- * Into the STDIO, MCH65, ..., and ROM directories, copy the corresponding source archive (for example, copy *stdio.arc* into the STDIO directory, *mch65.arc* into MCH65, and so on).

Extract the files from each archive using *arcv*, and then delete the archive.

Each of these directories contains the source and object modules generated from the corresponding source archive file. For example, the source files in STDIO were extracted

from the *stdio.arc* source archive file by the *arcv* program.

- * Into XFER, TTY, and CONFIG, copy the corresponding source archive (*xfer.arc* into XFER, etc). *xfer* transfers files between computers; *tty* is a terminal emulator; and *config* is used to define device attributes for programs generated with Aztec C65 for the Apple //. These programs are not absolutely necessary for the development of programs with Aztec CG65, and in fact you will probably have to modify them for use with your system, but they can be very useful.
- * Into the WORK directory, copy the *exmpl.c* sample C program. Later in this chapter, we are going to lead you through the steps to convert this program to Intel hex code.

2. Creating Object Module Libraries

The functions that are provided with Aztec CG65 are in source form. Before you can create an executable program using CG65, you must compile and assemble the functions and generate object module libraries that contain them, after first making any needed modifications. For more information, see the Library Generation chapter.

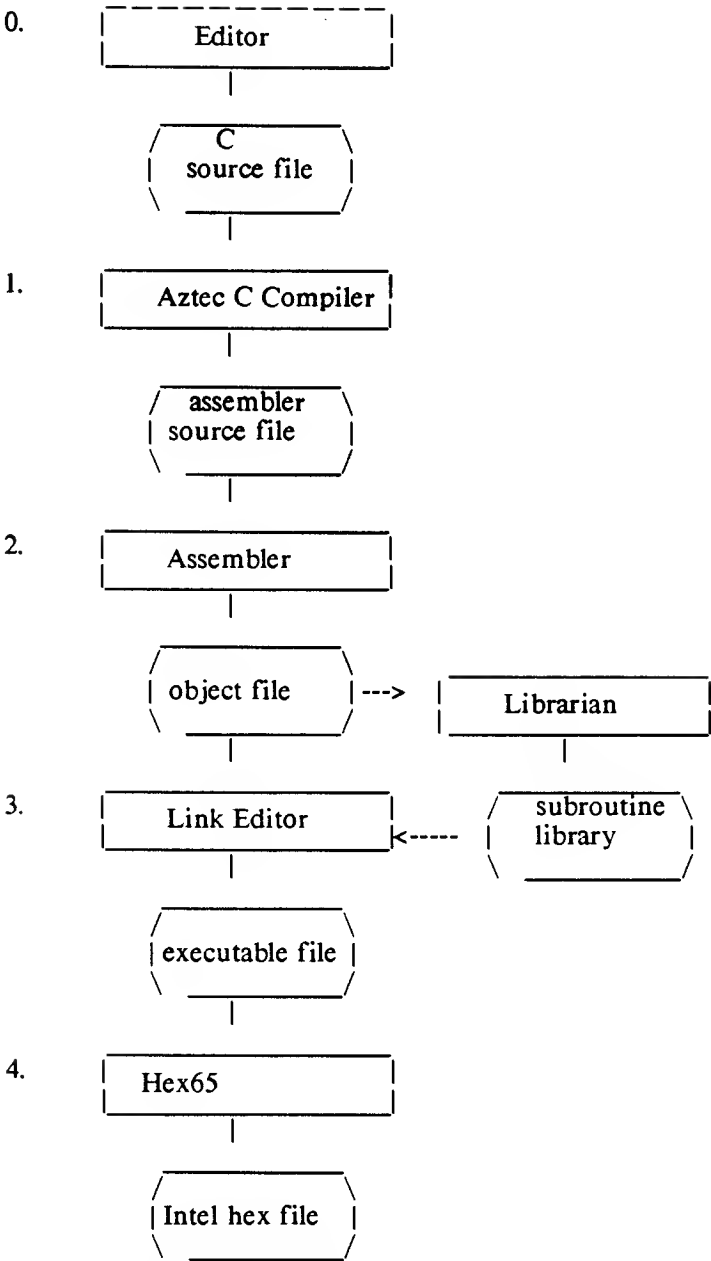


Figure 1: Program Development with Aztec CG65

3. Translating a program into Intel hex code

In this section we will lead you through the steps necessary to translate the sample C program whose source is in *exmpl.c* into Intel hex code. For a diagram of this procedure, see figure 1.

This program will be created so that it can reside in a system whose RAM occupies the bottom part of the memory space and whose ROM occupies the top part. In particular:

- * The program's data will be in RAM, beginning at address 0x200, thus leaving the first two pages of the memory space free for the usual page 0 and page 1 purposes;
- * The program's code will be in ROM, beginning at address 0xe000.
- * The 65xx power-up and interrupt vectors will be in ROM and will point to routines in the generated program.

3.1 Step 0: Create the Source Program

The first step to creating a C program is, of course, to create a disk file containing its source. This step isn't needed for this demonstration, since the source code already exists in the file *exmpl.c*.

For your own programs, you can create the C source using any text editor.

3.2 Steps 1 and 2: Compile and Assemble

To compile and assemble a C module, you must first decide which compiler and assembler you are going to use. For this example, we will assume that you are going to use the ones that generate native 65xx/65C02 code, *cg65* and *as65*. Later in this chapter we describe the compilers and assemblers that are provided with Aztec CG65.

Next, you must decide what zero-page locations you want the compiler-generated code to use. For this example, we will use the locations that are suitable for programs that are going to run on an Apple //. Later in this chapter we describe in more detail a program's use of the zero page.

Finally, having made the above decisions, you can compile and assemble *exmpl.c* by entering the following command:

```
cg65 +g0,8,80,10 exmpl.c
```

This first starts the *cg65* compiler, which translates the C source that's in *exmpl.c* into assembly language source. When done, *cg65* starts the *as65* assembler. *as65* assembles the assembly language source for the sample program, translating it into object code and writing the object code to the file *exmpl.r* in the current directory. When done, *as65* deletes the file that contains the assembly language source, since it is no longer needed.

The `+g0,8,80,10` argument tells the compiler about the generated code's use of the 65xx zero page. It says that the stack, temporary, and register areas begin at locations 0, 8, and 0x80, respectively; and the register area is 0x10 bytes long. This is discussed in more detail below.

3.3 Step 3: Link

The object code version of the *exmpl* program must next be linked to needed functions that are in the *c.lib* library of object modules and converted into a memory image.

Before entering this command, you must set the CLIB65 environment variable, to define the directory that contains the object module libraries. For example, on PC DOS, if the libraries are in *e:\cg65\lib*, the command to define CLIB65 is

```
set CLIB65=e:\cg65\lib\
```

Note the terminating slash: this is usually required, because of the way the linker builds the complete name of a library that is partially identified using the linker's *-l* option. This is described below.

The command to link the sample program is:

```
ln65 -t -b 200 -d 200 -c e000 exmpl.r -lc
```

There's a lot of parameters to this command, so let's go through them, one at a time:

3.3.1 The symbol table file and the *-T* option

The *-T* option tells the linker to write the program's symbol table information to the file *exmpl.sym*; this symbol table is needed by *hex65*, which converts the output of the linker into Intel hex code.

3.3.2 Segment addresses and the *-B*, *-D*, *-U*, and *-C* options

As you recall, we want the program's data to begin at 0x200 and its code at 0xe000. We tell this to the linker using the segment specification options: *-B*, *-D*, *-U*, and *-C*.

The *-B 200* option tells the linker that the program's "base address", that is, the address at which the linker-generated memory image can begin to be loaded into memory, is 0x200.

The *-D 200* option tells the linker that the program's initialized data is to begin at location 0x200. The linker organizes a program's data into two segments: its initialized data segment contains those of the program's global and static variables that are assigned an initial value (e.g. *static int var=1*); and its uninitialized data segment contains the program's other global and static variables. Just as the linker supports an option that tells it where to put the program's initialized data, it also supports a *-U* option, that tells it where to put the program's uninitialized data. When the *-U* option isn't used, the linker places uninitialized data immediately above the initialized data.

The `-C e000` option tells the linker that the program's code is to begin at location `0xe000`. Just as the linker groups all of a program's initialized and uninitialized data into segments, it groups all of a program's code into a code segment. The `-C` option defines the starting address of this segment.

On a 65xx, the top 6 bytes of the memory space contain vectors to the power-up and interrupt routines, and the first 512 bytes of memory contain dynamically-changing information. Because of this, most 65xx ROM systems have their ROM at the top of the memory space and their RAM at the bottom. The linker has default values for a program's base address and the beginning addresses of its segments, as follows:

- * The base address default to `0x800`;
- * The code segment begins three bytes past the base address;
- * The initialized data segment begins immediately after the code segment;
- * The uninitialized data segment begins immediately after the initialized data segment.

These default values are usually not appropriate for a ROM system, so you will usually use the linker's segment-specification options when generating a program that's going to be burned into ROM.

3.3.3 The input object module file and the memory image output file

The *exmpl.r* parameter explicitly tells the linker to include this module in the program that it's generating.

By default, the linker sends the output of the memory image it creates to a file whose name is derived from that of the first object module file that it encounters, by deleting the extension. Thus, the memory image for the above command is sent to the file *exmpl*. You can explicitly define the name of the memory image file using the linker's `-O` option.

3.3.4 Libraries and the `-L` option

The `-Lc` option tells the linker to search the *c.lib* library that's in the directory defined by the CLIB65 environment variable for needed functions.

As you can see, the `-L` option doesn't completely define the name of a library file; the linker generates the complete name by taking the letters that follow the `-L`, prepending them with the value of the CLIB65 environment variable, and appending the letters *.lib*. Thus, when CLIB65 has the value *e:\cg65\lib*, the `-Lc` option specifies the library whose complete file name is *e:\cg65\lib\c.lib*.

During the link step, the linker will search the libraries specified to it for modules containing needed functions; when such a module is found, the linker will include the module in the executable file it's

building.

All C programs need to be linked with *c.lib* (or its *cci*-compiled equivalent, *ci.lib*, as described below). This library contains the non-floating point functions which are defined in the functions chapter, *lib* of this manual. It also contains functions which are called by compiler-generated code.

If a program performs floating point operations, it must also be linked with a math library. The math library that you will use when getting familiar with Aztec C is *m.lib*. You can alternatively use its *cci*-compiled equivalent, *mi.lib*.

When a program is linked with a math library, that library must be specified before *c.lib*. For example, if *exmpl.c* performed floating point, the following would link it:

```
ln65 exmpl.r -lm -lc
```

3.4 Step 4: Convert to Intel hex code

The next step is to convert the memory image generated by the linker into Intel hex code. This is done with the following command:

```
hex65 exmpl
```

This command causes *hex65* to translate the program's memory image into Intel hex code. When this code is fed into a ROM programmer, the resulting ROM code will contain the program's code segment, a copy of its initialized data segment immediately following the code, and the power-up and reset vectors up at the top of memory.

Note: when the ROM system is started, its RAM contains random values, and the Aztec startup routine sets up the initialized data segment that resides in RAM from the copy that's in ROM.

hex65 generates Intel hex records, named *exmpl.x00*, *exmpl.x01*, and so on, for each 2 kb section of memory, beginning with the program's code segment. Thus, *exmpl.x00* contains the records for 0xe000-0xe800, *exmpl.x01* contains the records for 0xe800-0xf000, and so on.

The last hex file generated by *hex65* will contain records to initialize the nmi, reset, and irq vectors at the top of the 65xx address space. With the supplied software, these vectors point to locations in *rom.a65*. you can modify the software so that the vectors point to your own handlers.

If the ROM corresponding to the last hex file generated to hold the program's code and copy of its initialized data isn't the section of ROM that would be at the top of the 65xx memory space, *hex65* will output a separate file containing just those records needed to initialize the vectors in this last ROM. The extension on this file will indicate its sequence in the set of ROM chips needed to fill the memory space

from the beginning of the program's code to the top of memory; for example, if two 2 kb ROMs were sufficient to hold the program's code and copy of its initialized data, then the code and data would be in *exmpl.x00* and *exmpl.x01*, and the vectors would be in the file *exmpl.x03*.

There are several additional features of *hex65*. For example, *hex65* assumes that the size of each ROM is 2 kb long; using the *-P* option, you can explicitly define the size of each ROM. And by default, *hex65* generates the Intel hex records that set up these vectors; you can tell *hex65* not to generate these vector-initializing records. For a detailed description of *hex65*, see the Utility Programs chapter.

4. Special features of Aztec CG65

That concludes our step-by-step, cookbook introduction to Aztec CG65. In the following paragraphs, we want to describe two special features of Aztec CG65: its ability to generate either 65xx code or pseudo code, and the feature that allows you to define the locations within the zero page that generated programs will use.

4.1 Native Code vs. Pseudo Code

Aztec CG65 comes with two compilers and two assemblers: The *cg65* compiler and *as65* assembler, which together generate native machine code; and the *cci* compiler and *asi* assembler, which together generate pseudo code that must be interpreted.

There are advantages and disadvantages to using each compiler/assembler pair:

- * Code generated by *cg65* and *as65* is fast but large;
- * Code generated by *cci* and *asi* is small but slow.

Thus, when you are going to create an executable program, you must decide which compiler/assembler pair to use. We recommend that you first use *cg65* and *as65*. If it gets too large, use *cci* and *asi*. If neither of these alternatives is acceptable, with a native code version being too large and an interpreted version being too slow, you can divide the program into modules, compiling and assembling some of them into native code, the rest into interpreted code, and linking them all into a single executable program.

4.1.1 Native code and pseudo code libraries

Aztec CG65 provides "makefiles" with which you can generate two versions of each library: one whose modules are compiled with *cg65*, the other whose modules are compiled with *cci*. These libraries are:

<i>c.lib</i>	General purpose functions (<i>cg65</i> -compiled);
<i>ci.lib</i>	General purpose functions (<i>cci</i> -compiled);
<i>m.lib</i>	Floating point functions (<i>cg65</i> -compiled);
<i>mi.lib</i>	Floating point functions (<i>cci</i> -compiled).

As always, you can freely intermix *cg65*-compiled modules with *cci*-compiled modules, even when some of the modules come from one library or another.

4.2 Zero-page usage

The first 256 bytes of memory on a 65xx-based system are known as the "zero page", and are used differently by each system. Code generated by the Aztec CG65 compilers also makes use of the zero page, for storing variables. In order to allow CG65-generated code to be used on any 65xx-based system, the Aztec CG65 compilers group the zero-page variables used by generated code into three areas and allow you to define the location of these areas.

One area, which is 8 bytes long, contains the pseudo stack and frame pointers, and, if the program contains a *cci*-compiled module, the pseudo code interpreter's program counter.

Another area, which is 24 bytes long, contains five temporary registers, each of which is four bytes long.

The last area contains a program's register variables, and its size is specified by you when you compile a program. Thus, if your system uses most of the zero page, you can specify that your program uses few, or no, register variables. If your system has extra space in the zero page, you can fill it with register variables, thereby increasing the performance of your programs.

For example, the following table lists the starting addresses of the three areas and the size of the register variable area on the Apple //, Commodore 64, and the Atari 400/800. All values are in hexadecimal.

	Apple //	C-64	Atari
stack area addr	0	2	E0
temporary area addr	8	A	E8
Register var area addr	80	30	D4
Reg var area size	10	8	6

The location of these zero page locations are defined in two ways: with the *cg65* compiler's *+G* option, and in the assembly language file *zpage.h*:

4.2.1 Zero page usage of *cg65*-compiled modules

The *cg65* compiler's *+G* option defines the zero page usage of *cg65*-compiled, C language modules. For example, the following command compiles *hello.c* for use on the Commodore 64:

```
cg65 +g2,A,30,8 hello.c
```

4.2.2 Zero page usage of assembly language modules

The assembly language file *zpage.h* defines the zero page usage of assembly language modules. Normally, you will create a *zpage.h* file

early in your development cycle, before you create your libraries, since this file is included in the assembly of many of the library's assembly language modules. A version of *zpage.h* is supplied with Aztec CG65, and you can customize it for use with your system.

4.2.3 Zero page usage of *cci*-compiled modules

zpage.h also indirectly defines the zero page usage of *cci*-compiled, C language modules. The reasons for this are (1) the pseudo code interpreter, which executes *cci*-generated pseudo code, is an assembly language module that accesses zero page locations on behalf of a *cci*-compiled module, and (2) the locations of these zero page locations are defined by the *zpage.h* with which the interpreter is assembled.

cci itself produces machine-independent code; the same *cci*-generated object module can be executed on different 65xx systems, just by linking it with different object module versions of the interpreter, each of which has been generated by assembling the interpreter together with a *zpage.h* that defines the zero-page usage of the target system.

5. Where to go from here

In this chapter, we've just begun to describe the features of Aztec CG65.

One chapter that you must read is the Library Generation chapter, which discusses the generation of object module libraries from the source that comes with Aztec CG65.

We encourage you to use the *make* program-maintenance program to generate libraries, if such a program is available for your host system. To provide this encouragement, Aztec CG65 provides "makefiles" that can be used by UNIX-compatible *make* programs. If your host system is one, such as PCDOS, that doesn't have its own *make* program, and if the Aztec *make* is available for your system, it will be included in your Aztec CG65 package. A description of the Aztec *make* program is in the Utility Programs chapter.

For more information on the sections of a program, see the Program Organization section of the Technical Information chapter, and the section of the Linker chapter that discusses the segment specification options.

The *hex65* program supports several options that haven't been discussed in this introduction. For a complete description of this program see the Utility Programs chapter.

The Technical Information chapter contains information on several interesting topics, including the writing of assembly language functions, the pseudo stack, and object code format.

You should also read the Compiler, Assembler, and Linker chapters, to become familiar with all the options that these programs provide.

THE COMPILERS

Chapter Contents

The compilers	cc
1. Operating Instructions	3
1.1 The C Source File	3
1.2 The Output Files	4
1.3 Searching for <i>#include</i> files	6
2. Compiler Options	7
2.1 Summary of Options	7
2.2 Description of Options	9
3. Programmer Information	14
3.1 Supported Language Features	14
3.2 Structure Assignment	14
3.3 Line Continuation	14
3.4 The <i>void</i> Data Type	14
3.5 Special Symbols	15
3.6 String Merging	15
3.7 Long Names	16
3.8 Reserved Words	16
3.9 Global Variables	16
3.10 Data Formats	17
3.11 Floating Point Exceptions	18
3.11 Register Variables	20
3.12 In-line Assembly Language Code	20
3.13 Writing Machine-Independent Code	21
4. Error Processing	23

The Compilers

This chapter describes *cg65* and *cci*, the Aztec C compilers for the 65xx and 65C02 microprocessors. It is not intended to be a complete guide to the C language; for that, you must consult other texts. One such text is *The C Programming Language*, by Kernighan and Ritchie. The compilers were implemented according to the language description in the Kernighan and Ritchie book.

cg65 translates C source code into native 6502 assembly language source code. *cci* translates C source code into assembly language source for a "pseudo machine"; in an executable program, *cci*-compiled code must be interpreted by a special Aztec C routine.

This description of the compilers is divided into four subsections, which describe how to use the compilers, compiler options, information related to the writing of programs, and error processing.

To the operator and programmer, the two compilers are very similar. In the discussion that follows, we will use the name *cg65* when describing features that are common to both compilers. Where differences exist, we will say so.

1. Compiler Operating Instructions

cg65 is invoked by a command of the form:

```
cg65 [-options] filename.c
```

where [-options] specify optional parameters, and *filename.c* is the name of the file containing the C source program. Options can appear either before or after the name of the C source file.

The compiler reads C source statements from the input file, translates them to assembly language source, and writes the result to another file.

Upon completion, the compiler by default activates the *as65* assembler (*cci* by default starts the *asi* assembler). The assembler translates the assembly language source to relocatable object code, writes the result to another file, and deletes the assembly language source file. The *-A* option tells the compiler not to start the assembler.

1.1 The C source file

The extension on the source file name is optional. If not specified, it's assumed to be *.c*. For example, with the following command, the compiler will assume the file name is *text.c*:

cg65 text

The compiler will append *.c* to the source file name only if it doesn't find a period in the file name. So if the name of the source file really doesn't have an extension, you must compile it like this:

cg65 filename.

The period in the name prevents the compiler from appending *.c* to the name.

1.2 The output files

1.2.1 Creating an object code file

Normally, when you compile a C program you are interested in the relocatable object code for the program, and not in its assembly language source. Because of this, the compiler by default writes the assembly language source for a C program to an intermediate file and then automatically starts the assembler. The assembler then translates the assembly language source to relocatable object code, writes this code to a file, and erases the intermediate file.

By default, the object code generated by a *cg65*-started assembly is sent to a file whose name is derived from that of the file containing the C source by changing its extension to *.r* (the default extension for a *cci*-started assembly is *.i*). This file is placed in the directory that contains the C source file. For example, if the compiler is started with the command

cg65 prog.c

the file *prog.r* will be created, containing the relocatable object code for the program.

The name of the file containing the object code created by a compiler-started assembler can also be explicitly specified when the compiler is started, using the compiler's *-O* option. For example, the command

cg65 -O myobj.rel prog.c

compiles and assembles the C source that's in the file *prog.c*, writing the object code to the file *myobj.rel*.

When it's going to automatically start the assembler, the compiler by default writes the assembly language source to a temporary file named *ctmpxxx.xxx*, where the x's are replaced by digits in such a way that the name becomes unique. This temporary file is placed in the directory specified by the environment variable *CCTEMP*. If this variable doesn't exist, the file is placed in the current directory.

When *CCTEMP* exists, the fully-qualified name of the temporary file is generated by simply prefixing its value to the *ctmpxxx.xxx* name. For example if *CCTEMP* has the value

/RAM/TEMP/

then the temporary file is placed in the TEMP directory on the RAM volume.

For a description on the setting of environment variables, see your operating system manual.

If you are interested in the assembly language source, but still want the compiler to start the assembler, specify the option `-T` when you start the compiler. This will cause the compiler to (1) send the assembly language source to a file whose name is derived from that of the file containing the C source by changing its extension to `.asm` and (2) include the C source statements as comments in the assembly language source. For example, the command

```
cg65 -T prog.c
```

compiles and assembles *prog.c*, creating the files *prog.asm* and *prog.r*.

1.2.2 Creating just an assembly language file

There are some programs for which you don't want the compiler to automatically start the assembler. For example, you may want to modify the assembly language generated by the compiler for a particular program. In such cases, you can use the compiler's `-A` option to prevent the compiler from starting the assembler.

When you compile a program using the `-A` option, you can tell the compiler the name and location of the file to which it should write the assembly language source, using the `-O` option.

If you don't use the `-O` option but do use the `-A` option, the compiler will send the assembly language source to a file whose name is derived from that of the C source file by changing the extension to `.asm`, placing this file in the same directory as the one that contains the C source file. For example, the command

```
cg65 -A prog.c
```

compiles but doesn't assemble the C source that's in *prog.c*, sending the assembly language source to *prog.asm*.

As another example, the command

```
cg65 -A -O temp.a65 prog.c
```

compiles but doesn't assemble the C source that's in *prog.c*, sending the assembly language source to the file *temp.a65*.

When the `-A` option is used, the option `-T` causes the compiler to include the C source statements as comments in the assembly language source.

1.3 Searching for *#include* files

You can make the compiler search for *#include* files in a sequence of directories, thus allowing source files and *#include* files to be contained in different directories.

Directories can be specified with the *-I* compiler option, and with the INCL65 environment variable. The compiler itself also selects a few areas to search. The maximum number of searched areas is eight.

If the file name in the *#include* statement specifies a directory, just that directory is searched.

1.3.1 The *-I* option.

A *-I* option defines a single directory to be searched. The area descriptor follows the *-I*, with no intervening blanks. For example, the following *-I* option tells the compiler to search the */ram/include* directory:

```
-I/ram/include
```

1.3.2 The INCL65 environment variable.

The INCL65 environment variable also defines a directory to be searched for *#include* files. The value of this variable is the name of the directory to be searched.

The command that is used to set environment variables varies from system to system. For example, on PC DOS the following command sets INCL65 so that the directory *\CG65\INCLUDE* is searched for include files:

```
set INCL65=\CG65\INCLUDE
```

For a description of the command that's used on your system to set environment variables, see your operating system manual.

1.3.3 The search order for include files

Directories are searched in the following order:

1. If the *#include* statement delimited the file name with the double quote character, *"*, the current directory is automatically searched. If delimited by angle brackets, *<* and *>*, this area isn't automatically searched.
2. The directories defined in *-I* options are searched, in the order listed on the command line.
3. The directory defined in the INCL65 environment variable is searched.

2. Compiler Options

There are two types of options in Aztec C compilers: machine independent and machine dependent. The machine-independent options are provided on all Aztec C compilers. They are identified by a leading minus sign.

The Aztec C compiler for each target system has its own, machine-dependent, options. Such options are identified by a leading plus sign.

The following paragraphs first summarize the compiler options and then describe them in detail.

2.1 Summary of options

2.1.1 Machine-independent Options

- A Don't start the assembler when compilation is done.
- Dsymbol[=value] Define a symbol to the preprocessor.
- Idir Search the directory named *dir* for #include files.
- O file Send output to *file*.
- S Don't print warning messages.
- T Include C source statements in the assembly code output as comments. Each source statement appears before the assembly code it generates.
- B Don't pause after every fifth error to ask if the compiler should continue. See the Errors subsection for details.
- Enum Use an expression table having *num* entries.
- Lnum Use a local symbol table having *num* entries.
- Ynum Use a case table having *num* entries.
- Znum Use a literal table having *num* bytes.

2.1.2 Special Options for the 65xx Compilers

- +C Generate 65C02 code (*cg65* only).
- +B Don't generate the statement "public .begin".
- +L Turn automatic variables into statics (*cg65* only).
- +Gstk,tmp,reg,siz
(*cg65* only). Define zero-page locations for *cg65*-compiled modules: stack area begins at *stk*, temporary register area at *tmp*, register variable area begins at *reg* and is *siz* bytes long. The values are in hex. The zero page locations used by *cci*-compiled modules are

defined in *zpage.h*, when the pseudo code interpreter is assembled.

2.2 Detailed description of the options

2.2.1 Machine-independent options

The -D Option (Define a macro)

The `-D` option defines a symbol in the same way as the preprocessor directive, `#define`. Its usage is as follows:

```
cg65 -Dmacro[=text] prog.c
```

For example,

```
cg65 -DMAXLEN=1000 prog.c
```

is equivalent to inserting the following line at the beginning of the program:

```
#define MAXLEN 1000
```

Since the `-D` option causes a symbol to be defined for the preprocessor, this can be used in conjunction with the preprocessor directive, `#ifdef`, to selectively include code in a compilation. A common example is the following code:

```
#ifdef DEBUG
    printf("value: %d\n", i);
#endif
```

This debugging code would be included in the compiled source by the following command:

```
cg65 -dDEBUG program.c
```

When no substitution text is specified, the symbol is defined to have the numerical value 1.

The -I Option (Include another source file)

The `-I` option causes the compiler to search in a specified directory for files included in the source code. The name of the directory immediately follows the `-I`, with no intervening spaces. For more details, see the Compiler Operating Instructions, above.

The -S Option (Be Silent)

The compiler considers some errors to be genuine errors and others to be possible errors. For the first type of error, the compiler always generates an error message. For the second, it generates a warning message. The `-S` option causes the compiler to not print warning messages.

2.2.1.1 The Local Symbol Table and the -L Option

When the compiler begins processing a compound statement, such as the body of a function or the body of a *for* loop, it makes entries about the statement's local symbols in the local symbol table, and

removes the entries when it finishes processing the statement. If the table overflows, the compiler will display a message and stop.

By default, the local symbol table contains 40 entries. Each entry is 26 bytes long; thus by default the table contains 520 bytes.

You can explicitly define the number of entries in the local symbol table using the `-L` option. The number of entries immediately follows the `-L`, with no intervening spaces. For example, the following compilation will use a table of 75 entries, or almost 2000 bytes:

`cg65 -L75 program.c`

2.2.1.2 The Expression Table and the `-E` Option

The compiler uses the expression table to process an expression. When the compiler completes its processing of an expression, it frees all space in this table, thus making the entire table available for the processing of the next expression. If the expression table overflows, the compiler will generate error number 36, "no more expression space", and halt.

By default, the expression table contains 80 entries. Each entry is 14 bytes long; thus by default the table contains 1120 bytes.

You can explicitly define the number of entries in the expression table using the `-E` option. The number of entries immediately follows the `-E`, with no intervening spaces. For example, the following compilation will use a table of 20 entries:

`cg65 -E20 program.c`

2.2.1.3 The Case Table and the `-Y` Option

The compiler uses the case table to process a switch statement, making entries in the table for the statement's cases. When it completes its processing of a switch statement, it frees up the entries for that switch. If this table overflows, the compiler will display error 76 and halt.

For example, the following will use a maximum of four entries in the case table:


```

switch (a) {
case 0:                /* one */
    a += 1;
    break;
case 1:                /* two */
    switch (x) {
    case 'a':          /* three */
        func1 (a);
        break;
    case 'b':          /* four */
        func2 (b);
        break;
    }                  /* release the last two */
    a = 5;
case 3:                /* total ends at three */
    func2 (a);
    break;
}

```

By default, the table contains 100 entries. Each entry is four bytes long; thus by default, the table occupies 400 bytes.

You can explicitly define the number of entries in the case table using the compiler's `-Y` option. The number of entries immediately follows the `-Y`, with no intervening spaces. For example, the following compilation uses a case table having 50 entries:

cg65 -Y50 file

2.2.1.4 The String Table and the `-Z` Option

When the compiler encounters a "literal" (that is, a character string), it places the string in the literal table. If this table overflows, the compiler will display error 2, "string space exhausted", and halt.

By default, the literal table contains 2000 bytes.

You can explicitly define the number of bytes in this table using the compiler's `-Z` option. The number of bytes immediately follows the `-Z`, with no intervening spaces. For example, the following command will reserve 3000 bytes for the string table:

cg65 -Z3000 file

2.2.1.5 The Macro/Global Symbol Table

The compiler stores information about a program's macros and global symbols in the Macro/Global Symbol Table. This table is located in memory above all the other tables used by the compiler. Its size is set after all the other tables have been set, and hence can't be set by you. If this table overflows, the compiler will display the message "Out of Memory!" and halt. You must recompile, using smaller sizes for the other tables.

2.2.2 65xx Options

2.2.2.1 The +G Option (Define zero page usage for *cg65*-compiled modules)

The +G option defines the zero page locations that will be used by *cg65*-generated code. The option has the form

+Gsaddr,taddr,uaddr,ucnt

where

<i>saddr</i>	Starting address, in hex, of the stack area. This area is 8 bytes long and by default begins at location 0.
<i>taddr</i>	Starting address, in hex, of the temporary register area. This area is 24 bytes long and by default begins at location 8.
<i>uaddr</i>	Starting address, in hex, of the user register area. The size of this area is two times the value that is specified for the +G option's <i>ucnt</i> parameter. By default, this area begins at location 0x80.
<i>ucnt</i>	The number of bytes in the register variable area, in hex. By default, this area is 16 bytes long; ie, contains space for eight register variables.

No spaces are allowed in the +G option.

The default values for unspecified +G fields are those used for Apple // programs.

As an example of the use of this option, the following command compiles the "hello, world" program for use on a Commodore 64, which uses *saddr*=2, *taddr*=0xa, *uaddr*=0x30, and *ucnt*=8:

```
cg65 +g2,a,30,8 hello.c
```

The +G option is not used by the *cci* compiler. The zero page usage of *cci*-compiled modules is defined when the pseudo code interpreter *.interp* is assembled.

2.2.2.2 The +C Option (Generate 65C02 code - *cg65* only)

The +C option causes *cg65* to generate assembler source for a 65C02 processor. If this option isn't used, *cg65* will generate code for a 6502 processor.

2.2.2.3 The +B Option (Don't generate reference to *.begin*)

Normally when compiling a module, the compilers generate a reference to the entry point named *.begin*. Then when the module is linked into a program, the reference causes the linker to include in the program the library module that contains *.begin*.

The +B option prevents the compilers from generating this reference.

For example, if you want to provide your own entry point for a program, and its name isn't *.begin*, you should compile the program's modules with the *+B* option. If you don't, then the program will be bigger than necessary, since it will contain your entry point module and the standard entry point module. In addition, the linker by default sets at the program's base address a jump instruction to the program's entry point; if it finds entry points in several modules, it will set the jump to the last one encountered.

2.2.2.4 The *+L* Option (Turn Autos into Statics - *cg65* only)

The *+L* option causes the compiler to change the class of variables whose class is automatic to static. This can cause a significant increase in execution speed, since it is faster to address static variables, which are directly addressable, than automatic variables, which are on the stack and must be indirectly addressed.

Automatic variables that are declared using the *auto* keyword, (for example *auto int i*), aren't affected by the *+L* option: they will remain automatic.

Also, if a register is available for an automatic variable that is declared using the *register* keyword (for example, *register int i*), the variable will be placed in a register and will not be turned into a static. If a register is not available, however, such a variable will be turned into a static variable.

Like any other static data, an auto-turned-static is initialized to zero before the program begins.

A function that recursively calls itself may not work correctly when it is compiled with the *+L* option. For example, the following program will print 1 when compiled without the *+L* option, and 100 when compiled with the *+L* option:

```
main()
{
    printf("%d", qtest());
}

qtest()
{
    int i;
    if (++i < 100)
        qtest(i);
    return (i);
}
```

3. Writing programs

The previous sections of this description of the compiler discussed operational features of the compiler; that is, presented information that an operator would use to compile a C program. In this section, we want to present information of interest to those who are actually writing programs.

3.1 Supported Language Features

Aztec C supports the entire C language as defined in *The C Programming Language* by Kernighan and Ritchie. This now includes the bit field data type.

The following paragraphs describe features of the standard C language that are supported by Aztec C but aren't described in the K & R text.

3.2 Structure assignment

Aztec C supports structure assignment. With this feature, a program can cause one structure to be copied into another using the assignment operator.

For example, if *s1* and *s2* are structures of the same type, you can say:

```
s1 = s2;
```

thus causing the contents of structure *s1* to be copied into structure *s2*.

Unlike other operators, the assignment operator doesn't have a value when it's used to copy a structure. Thus, you can't say things like "*a* = *b* = *c*", or "*(a=b).fld*" when *a*, *b*, and *c* are structures.

3.3 Line continuation

If the compiler finds a source line whose last character is a backslash, \, it will consider the following line to be part of the current line, without the backslash. For example, the following statements define a character array containing the string "abcdef":

```
char array[]="ab\  
cd\  
ef";
```

3.4 The void data type

Functions that don't return a value can be declared to return a *void*. This provides a safety check on the use of such functions. If a *void* function attempts to return a value, or if a function tries to use the value returned by a *void* function, the compiler will generate an error message.

Variables can be declared to point to a *void*, and functions can be declared as returning a pointer to a *void*.

When an assignment of one pointer to another is made, the compiler usually wants both pointers to point at the same type of object; otherwise, it will issue a warning message. However, a pointer to an object of type *void* can be assigned to, and can itself be assigned to, a pointer to an object of any type without causing the compiler to complain.

That is, the compiler will generate a warning message for the assignment statement in the following program:

```
main()
{
    char *cp;
    int *ip;
    ip = cp;
}
```

The compiler won't complain about the following program:

```
main()
{
    char *cp;
    void *getbuf();
    cp = getbuf();
}
```

3.5 Special symbols

Aztec C supports the following symbols:

___FILE___	Name of the file being compiled. This is a character string.
___LINE___	Number of the line currently being compiled. This is an integer.
___FUNC___	Name of the function currently being compiled. This is a character string.

In case you can't tell, these symbols begin and end with two underscore characters.

For example,

```
printf("file= %s\n", ___FILE___);
printf("line= %d\n", ___LINE___);
printf("func=%s\n", ___FUNC___);
```

3.6 String merging

The compiler will merge adjacent character strings. For example,

```
printf("file=" ___FILE___ " line= %d func= " ___FUNC___,
      ___LINE___);
```

3.7 Long names

Symbol names are significant to 31 characters. This includes external symbols, which are significant to 31 characters throughout assembly and linkage.

3.8 Reserved words

const, *signed*, and *volatile* are reserved keywords, and must not be used as symbol names in your programs.

3.9 Global variables

Aztec C supports the rule of the standard C language regarding global variables that are to be accessed by several modules. This rule requires that in the modules that want to access such a variable, exactly one module declare it without the *extern* keyword and all others declare it with the *extern* keyword.

Previous versions of Aztec C did not strictly enforce this rule. In these versions, the following modified version of the rule was enforced:

- * multiple modules could declare the same variable, with the *extern* keyword being optional;
- * when several modules declared a variable without using the *extern* keyword, the amount of space reserved for the variable was set to the largest size specified by the various declarations;
- * when one module declared a variable using the *extern* keyword, at least one other module must have declared the variable without using the *extern* keyword;
- * at most one module could specify an initial value for a global variable;
- * when a module specified an initial value for a global variable, the amount of storage reserved for the variable was set to the amount specified in the declaration that specified an initial value, regardless of the amounts specified in the other declarations.

In order to (1) enforce the standard C rule regarding global variables and (2) provide compatibility with previous versions of Aztec C, the current Aztec linker will generate code consistent with the previous versions, but will by default generate a "multiply defined symbol" message when multiple modules are found that declare a global variable without the *extern* keyword. The *-M* linker option can be used to cause the linker to treat global variables just as they were in previous versions of Aztec C; in this case, the "multiply defined symbol" message won't occur when several modules declare the same variable without the *extern* keyword, as long as no more than one specifies an initial value for the variable. If multiple modules declare an initial value for the same variable this message will be issued,

regardless of the use of the *-M* option.

Both previous and current versions of Aztec C prevent a global symbol from being both a variable name and a function name. When such a situation arises, the linker will issue the "multiply defined symbol" message, regardless of the use of the *-M* option.

3.10 Data formats

3.10.1 char

Variables of type *char* are one byte long, and can be signed or unsigned. By default, a *char* variable is unsigned.

When a signed *char* variable is used in an expression, it's converted to a 16-bit integer by propagating the most significant bit. Thus, a *char* variable whose value is between 128 and 255 will appear to be a negative number if used in an expression.

When an unsigned *char* variable is used in an expression, it's converted to a 16-bit integer in the range 0 to 255.

A character in a *char* is in ASCII format.

3.10.2 pointer

Pointer variables are two bytes long.

3.10.3 int, short

Variables of type *short* and *int* are two bytes long, and can be signed or unsigned.

A negative value is stored in two's complement format. A -2 stored at location 100 would look like:

<i>location</i>	<i>contents in hex</i>
100	FE
101	FF

3.10.4 long

Variables of type *long* occupy four bytes, and can be signed or unsigned.

Negative values are stored in two's complement representation. Longs are stored sequentially with the least significant byte stored at the lowest memory address and the most significant byte at the highest memory address.

3.10.5 float

A *float* variable is represented internally by a sign flag, a base-256 exponent in excess-64 notation, and a three-character, base-256 fraction. All variables are normalized.

The variable is stored in a sequence of four bytes. The most significant bit of byte 0 contains the sign flag; 0 means it's positive, 1 negative.

The remaining seven bits of byte 0 contain the excess-64 exponent.

Bytes 1,2, and 3 contain the three-character mantissa, with the most significant character in byte 1 and the least in byte 3. The 'decimal point' is to the left of the most significant byte.

As an example, the internal representation of decimal 1.0 is 41 01 00 00.

3.10.6 Doubles

A floating point number of type *double* is represented internally by a sign flag, a **base-256** exponent in excess-64 notation, and a seven-character, base-256 fraction.

The variable is stored in a sequence of eight bytes. The most significant bit of byte 0 contains the sign flag; 0 means positive, 1 negative.

The excess-64 exponent is stored in the remaining seven bits of byte 0.

The seven-character, base-256 mantissa is stored in bytes 1 through 7, with the most significant character in byte 1, and the least in byte 7. The "decimal point" is to the left of the most significant character.

As an example, $(256^{**}3)*(1/256 + 2/256^{**}2)$ is represented by the following bytes: 43 01 02 00 00 00 00 00.

For accuracy, floating point operations are performed using mantissas which are 16 characters long. Before the value is returned to the user, it is rounded.

3.11 Floating Point Exceptions

When a C program requests that a floating point arithmetic operation be performed, a call will be made to functions in the floating point support software.

While performing the operation, these functions check for the occurrence of the floating point exception conditions; namely, overflow, underflow, and division by zero. On return to the caller, the global integer *fltterr* indicates whether an exception has occurred:

<i>fltterr</i>	<i>value returned</i>	<i>meaning</i>
0	computed value	no error has occurred
1	+/- 2.9e-157	underflow
2	+/- 5.2e151	overflow
3	+/- 5.2e151	division by zero

If the value of *flterr* is zero, no error occurred, and the value returned is the computed value of the operation. Otherwise, an error has occurred, and the value returned is arbitrary. The table lists the possible settings of *flterr*, and for each setting, the associated value returned and the meaning.

When a floating point exception occurs, in addition to returning an indicator in *flterr*, the floating point support routines will either log an error message to the console or call a user-specified function. The error message logged by the support routines define the type of error that has occurred (overflow, underflow, or division by zero) and the address, in hex, of the instruction in the user's program which follows the call to the support routines.

Following the error message or call to a user function, the floating point support routines return to the user's program which called the support routines.

To determine whether to log an error message itself or to call a user's function, the support routines check the first pointer in *Sysvec*, the global array of function pointers. If it contains zero (which it will, unless the user's program explicitly sets it), the support routines log a message; otherwise, the support routines call the function pointed at by this field.

A user's function for handling floating point exceptions can be written in C. The function can be of any type, since the support routines don't use the value returned by the user's function. The function has two parameters: the first, which is of type *int*, is a code identifying the type of exception which has occurred. The value 1 indicates underflow, 2 overflow, and 3 division by zero.

The second parameter passed to the user's exception-handling routine is a pointer to the instruction in the user's program which follows the call instruction to the floating point support routines. One way to use this parameter would be to declare it to be of type *int*. The user's routine could then convert it to a character string for printing in an error message.

The example below demonstrates how floating point errors can be trapped and reported. In *main*, a pointer in the *Sysvec* array is set to the routine, *usertrap*. If a floating point exception occurs during the execution of the program, this routine is called with the arguments described above. The error handling routine prints the appropriate error message, and returns to the floating point support routines.

```

#include <stdio.h>

main() {
    Sysvec[FLT__FAULT] = usertrap;
}

usertrap(errcode,addr)
int errcode,addr;
{
    char buff[4];
    switch (errcode) {
        case '1':
            printf("floating point underflow at %x\n",buff);
            break;
        case '2':
            printf("floating point overflow at %x\n",buff);
            break;
        case '3':
            printf("division by zero at %x\n", buff);
            break;
        default:
            printf("usertrap: invalid code %d \n", errcode);
            break;
    }
}

```

3.12 Register Variables

A *cg65*-compiled program can have up to eight register variables. A *cci*-compiled program can declare variables to be of type *register*, but the compiler will ignore the declaration.

3.13 In-Line Assembly Language Code

Assembly language source can be included in a C program, by surrounding the assembly language code with the preprocessor directives *#asm* and *#endasm*.

When the compiler encounters a *#asm* statement, it copies lines from the C source file to the assembly language file that it's generating, until it finds a *#endasm* statement. The *#asm* and *#endasm* statements are not copied.

While the compiler is copying assembly language source, it doesn't try to process or interpret the lines that it reads. In particular, it won't perform macro substitution.

A program that uses *#asm ...#endasm* must avoid placing in-line assembly code immediately following an *if* block; that is, it should avoid the following code:

```

    if (...){
        ...
    }
    #asm
    ...
    #endasm
    ...

```

The code generated by the compiler will test the condition and if false branch to the statement following the `#endasm` instead of to the beginning of the assembly language code. To have the compiler generate code that will branch to the beginning of the assembly language code, you must include a null statement between the end of the `if` block and the `asm` statement:

```

    if (...){
        ...
    }
    ;
    #asm
    ...
    #endasm
    ...

```

3.14 Writing machine-independent code

The Aztec family of C compilers are almost entirely compatible. The degree of compatibility of the Aztec C compilers with v7 C, system 3 C, system 5 C, and XENIX C is also extremely high. There are, however, some differences. The following paragraphs discuss things you should be aware of when writing C programs that will run in a variety of environments.

If you want to write C programs that will run on different machines, don't use bit fields or enumerated data types, and don't pass structures between functions. Some compilers support these features, and some don't.

3.14.1 Compatibility Between Aztec Products

Within releases, code can be easily moved from one implementation of Aztec C to another. Where release numbers differ (i.e. 1.06 and 2.0) code is upward compatible, but some changes may be needed to move code down to a lower numbered release. The downward compatibility problems can be eliminated by not using new features of the higher numbered releases.

3.14.2 Sign Extension For Character Variables

If the declaration of a *char* variable doesn't specify whether the variable is signed or unsigned, the code generated for some machines assumes that the variable is signed and others that it's unsigned. For

example, none of the 8 bit implementations of Aztec C sign extend characters used in arithmetic computations, whereas all 16 bit implementations do sign extend characters. This incompatibility can be corrected by declaring characters used in arithmetic computations as unsigned, or by AND'ing characters used in arithmetic expressions with 255 (0xff). For instance:

```
char a=129;
int b;
b = (a & 0xff) * 21;
```

3.14.3 The MPU... symbols

To simplify the task of writing programs that must have some system dependent code, each of the Aztec C compilers defines a symbol which identifies the machine on which the compiler-generated code will run. These symbols, and their corresponding processors, are:

<i>symbol</i>	<i>processor</i>
MPU68000	68000
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502
MPU8080	8080
MPUZ80	Z80

Only one of these symbols will be defined for a particular compiler.

For example, the following program fragment contains several machine-dependent blocks of code. When the program is compiled for execution on a particular processor, just one of these blocks will be compiled: the one containing code for that processor.

```
#ifdef MPU68000
    /* 68000 code */
#else
#ifdef MPU8086
    /* 8086 code */
#else
#ifdef MPU8080
    /* 8080 code */
#endif
#endif
#endif
```

4. Error checking

Compiler errors come in two varieties-- fatal and not fatal. Fatal errors cause the compiler to make a final statement and stop. Running out of memory and finding no input are examples of fatal errors. Both kinds of errors are described in the *Errors* chapter. The non-fatal sort are introduced below.

The compiler will report any errors it finds in the source file. It will first print out a line of code, followed by a line containing the up-arrow (caret) character. The up-arrow in this line indicates where the compiler was in the source line when it detected the error. The compiler will then display a line containing the following:

- * The name of the source file containing the line;
- * The number of the line within the file;
- * An error code;
- * The symbol which caused the error, when appropriate.

The error codes are defined and described in the *Errors* chapter.

The compiler writes error messages to its standard output. Thus, error messages normally go to the console, but they can be associated with another device or file by redirecting standard output in the usual manner. For example,

<code>cg65 prog</code>	errors sent to the console
<code>cg65 prog >outerr</code>	errors sent to the file <i>outerr</i>

The compiler normally pauses after every fifth error, and sends a message to its standard output asking if you want to continue. The compiler will continue only if you enter a line beginning with the character 'y'. If you don't want the compiler to pause in this manner, (if, for example, the compiler's standard output has been redirected to a file) specify the *-B* option when you start the compiler.

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error.

If errors arise at compile time, it is a general rule of thumb that the very first error should be corrected first. This may clear up some of the errors which follow.

The best way to attack an error is to first look up the meaning of the error code in the back of this manual. Some hints are given there as to what the problem might be. And you will find it easier to understand the error and the message if you know why the compiler produced that particular code. The error codes indicate what the compiler was doing when the error was found.

THE ASSEMBLERS

Chapter Contents

The Assemblers	as
1. Operating Instructions	3
1.1 The Source File	3
1.2 The Object Code File	4
1.3 Listing File	4
1.4 Searching for <i>instxt</i> Files	4
2. Assembler Options	5
3. Programmer information	5

The Assemblers

as65 and *asi* are relocating assemblers that translate an assembly language source program into relocatable object code. The two assemblers support different machines: *as65* accepts assembly language for a 6502 or 65c02; *asi* accepts assembly language for a "pseudo machine".

In an executable program, an *asi*-assembled module must be interpreted by a routine that is in the Aztec libraries.

An executable program can contain both modules that have been assembled with *as65* and modules that have been assembled with *asi*.

This description has three sections: the first describes how to operate the assembler; the second describes the assembler's options; and the third presents information of interest to those writing assembly language programs.

1. Operating Instructions

Operationally, the two assemblers are very similar. In the following paragraphs, we will use the name *as65* when referring to features that are common to both assemblers. When the two assemblers differ, we will say so.

as65 is started with a command line of the form

```
as65 [-options] prog.asm
```

where *[-options]* are optional parameters and *prog.asm* is the name of the file to be assembled. *as65* reads the source code from the specified file, translates it into object code, and writes the object code to another file.

1.1 The Source File

The extension on the source file name is optional. If not specified, it's assumed to be *.asm*. For example, with the following command, the compiler will assume that the file name is *test.asm*:

```
as65 test
```

as65 will append *.asm* to the source file name only if it doesn't find a period in the file name. So if the name of the source file really doesn't have an extension, you must compile it like this:

```
as65 filename.
```

The period tells the assembler not to append *.asm* to the name.

1.2 The Object File

By default, the name of the file to which *as65* writes object code is derived from the name of the source code file, by changing its extension to *.r* (or to *.i*, if *asi* is used). Also by default, the object code file is placed in the directory that contains the source code file. For example, the command

```
as65 test.asm
```

writes object code to the file *test.r* (or to *test.i*, if *asi* is used), placing this file in the current directory.

You can explicitly specify the name of the object code file, using the *-O* option. The name of the object code file follows the *-O*, with spaces between the *-O* and the file name. For example, the following command assembles *test.asm*, writing the object code to the file *prog.out*:

```
as -o prog.out test.asm
```

1.3 The Listing File

The *-L* option causes the assembler to create a file containing a listing of the program being assembled. The file is placed in the directory that contains the object file; its name is derived from that of the object file by changing the extension to *.lst*.

1.4 Searching for *instxt* files

The *instxt* directive tells *as65* to suspend assembly of one file and assemble another; when assembly of the second file is completed, assembly of the first continues.

You can make the assembler search for *instxt* files in a sequence of directories, thus allowing source files and *instxt* files to be in different directories.

Directories that are to be searched are defined just as for the compilers; that is, using the *-I* assembler option and the *INCL65* environment variable. Optionally, the compiler can also search the current directory.

Directory search for a particular *instxt* directive can be disabled by specifying a directory name in the directive. In this case, just the specified directory is searched.

1.4.1 The *-I* option

A *-I* option defines a single directory to be searched. The directory name follows the *-I*, with no intervening blanks. For example, the following *-I* option tells the assembler to search the */ram/include* directory:

-I/ram/include

1.4.2 The INCL65 environment variable.

The INCL65 environment variable defines a directory to be searched for *instxt* files. The value of this variable is the name of the directory to be searched.

The command that is used to set environment variables varies from system to system. For example, on PCDOS the following command sets INCL65 so that the directory \CG65\INCLUDE is searched for include files:

set INCL65=\CG65\INCLUDE

For a description of the command that's used on your system to set environment variables, see your operating system manual.

1.4.3 The search order

Directories are searched in the following order:

1. If the *instxt* directive delimited the file name with the double quote character, ", the current directory on the default drive is searched. If delimited by angle brackets, < and >, this directory isn't automatically searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directory defined in the INCL65 environment variable is searched.

2. Assembler Options

The assembler supports the following options:

<i>Option</i>	<i>Meaning</i>
-O <i>objname</i>	Send object code to <i>objname</i> .
-L	Generate listing.
-C	Disable assembly of 65C02 instructions. Not supported by <i>asi</i> .
-ZAP	Delete the source file after assembling it.

3. Programming Information

This section discusses the assembly language that is supported by *as65*. A description of the assembly language supported by *asi* is not available.

as65 supports the standard MOS Technology syntax: a program consists of a sequence of statements, each of which is in the standard MOS Tech form; and the assembler supports the MOS Tech mnemonics for the standard instructions. *as65* supports some of the MOS Tech directives and their mnemonics; it also supports others, as

defined below.

The following paragraphs define in more detail the language supported by *as65*.

3.1 Statement Syntax

[label] [opcode] [arguments] [[;]comment]

where the brackets "[...]" indicate an optional element.

3.2 Labels

A statement's label field defines a symbol to the assembler and assigns it a value. If present, the symbol name begins in column one. If a statement is not labeled, then column one must be a blank, tab, or asterisk. An asterisk denotes a comment line.

Normally, the symbol in a label field is assigned as its value the address at which the statement's code will be placed. However, the *equ* directive can be used to create a symbol and assign it some other value, such as a constant.

A label can contain up to 32 characters. Its first character must be an alphabetic character or one of the special characters ' ' or '.'. Its other characters can be alphabetic characters, digits, ' ', or '.'. A label followed by "#" is declared external.

The compilers place a ' ' character at the end of all labels that they generate.

3.3 Opcodes

The assembler supports the standard MOS Tech instruction mnemonics for both the 6502 and 65C02 processors. The directives it supports are defined below.

3.4 Arguments

A statement's arguments can specify a register, a memory location, or a constant.

A memory location can be referenced using any of the standard 6502 or 65C02 addressing modes, and using the standard MOS Tech syntax.

A memory location reference or a constant can be an expression containing any of the following operators:

*	multiply
/	divide
+	add
-	subtract
#	constant
=	constant
<	low byte of expression
>	high byte of expression

Expressions are evaluated from left to right with no precedence as to operator or parentheses.

3.5 Constants

The default base for numeric constants is decimal. Other bases are specified by the following prefixes or suffixes:

<i>Base</i>	<i>Prefix</i>	<i>Suffix</i>
2	%	b,B
8	@	o,O,q,Q
10	null,&	null
16	\$	h,H

A character constant consists of the character, preceded by a single quote. For example: 'A'.

3.6 Directives

The following paragraphs describe the directives that are supported by the assembler.

END

end

The *end* directive defines the end of the source statements.

CSEG

cseg

The *cseg* directive selects a module's code segment; information generated by statements that follow a *cseg* directive is placed in the module's code segment, until another segment-selection directive is encountered.

DSEG

dseg

The *dseg* directive selects a module's data segment; information generated by statements that follow a *dseg* directive is placed in the module's data segment, until another segment-selection directive is encountered.

EQU

symbol *equ* *<expr>*

The *equ* directive creates a symbol named *symbol* (if it doesn't already exist), and assigns it the value of the expression *expr*.

PUBLIC

public *<symbol>[,<symbol>...]*

The *public* directive identifies the specified symbols as having external scope. If a specified symbol was created in the within the module that's being assembled (by being defined in a statement's label field), this directive allows it to be accessed by other modules. If a symbol was not created within the module that's being assembled, this directive tells the assembler that the symbol was created and made public in another module.

BSS

bss *<symname>,<size>*

The *bss* directive creates a symbol named *symname* and reserves *size* bytes of space for it in the uninitialized data segment. The symbol cannot be accessed by other modules.

GLOBAL

global *<symnam>,<size>*

The *global* directive creates a symbol named *symnam* that other modules can access using the *global* and *public* directives.

If other modules create *symnam* using just the *global* directives, then *symnam* will be located in a program's uninitialized data area. In this case, the amount of space reserved in this area for *symnam* will equal the largest value specified by the *size* fields in the *global* statements that define *symnam*.

If other modules define *symnam* in a *public* statement, but none of them create *symnam* (by specifying it in a label field), then *symnam* will still be located in the uninitialized data segment and space will be reserved for it as defined above.

If one module both defines *symnam* using a *public* statement and creates the symbol by specifying it in a label field, then *symnam* will be located in the program's code or data segment and no space will be reserved for it in the uninitialized data segment.

ENTRY

entry *<symnam>*

The *entry* directive defines the symbol *symnam* as being a program's entry point.

When a program is linked, the linker normally places a jump instruction at the program's base address. If the linker finds a module containing an *entry* directive, it sets the target of the jump to the location that was specified in the last *entry* directive that it found; otherwise, it sets the target to the beginning of the program's code segment.

FCB

[*label*] *fcb* <*value*>[,<*value*>, <*value*> ...]

Each *value* in an *fcb* directive causes one or more bytes of memory to be allocated and then initialized to the specified value. The memory is allocated in the currently active segment (code or data, as defined by the last segment-selection directive).

FDB

[*label*] *fdb* <*value*>[,<*value*>, <*value*> ...]

The *fdb* directive is like *fcb*, except that each *value* causes a two-byte field of memory to be allocated and initialized.

FCC

[*label*] *fcc* "*string*"

The *fcc* directive allocates a field that has the same number of characters as are in *string*, and places *string* in it. The field is placed in the currently-active segment.

RMB

[*label*] *rmb* <*expr*>

The *rmb* directive reserves a field containing *expr* bytes in the currently-active segment. The contents of the field are not defined.

INSTXT

instxt <*file*>
instxt "*file*"
instxt / *file* /

The *instxt* directive causes the assembler to suspend assembly of the current source file and to assemble the source that's in *file*. When done, the assembler will continue assembling the original file.

The assembler can search for a file in several directories. If *file* is surrounded by quotes or slashes, the assembler will begin the search at the current directory; it will then search directories specified in the -I option and the INCL65 environment variable. If *file* is surrounded by <>, the assembler will search just the -I and INCL65 directories.

THE LINKER

Chapter Contents

The Linker ln

1. Introduction to linking 3

2. Using the Linker 7

3. Linker Options 9

The Linker

The *ln65* linker has two functions:

- * It ties together the pieces of a program which have been compiled and assembled separately;
- * It converts the linked pieces to a format which can be loaded and executed.

The pieces must have been created by the Manx assembler.

The first section of this chapter presents a brief introduction to linking and what the linker does. If you have had previous experience with linkage editors, you may wish to continue reading with the second section, entitled "Using the Linker." There you will find a concise description of the command format for the linker.

1. Introduction to linking

Relocatable Object Files

The object code produced by the assembler is "relocatable" because it can be loaded anywhere in memory. One task of the linker is to assign specific addresses to the parts of the program. This tells the operating system where to load the program when it is run.

Linking hello.r

It is very unusual for a C program to consist of a single, self-contained module. Let's consider a simple program which prints "hello, world" using the function, *printf*. The terminology here is precise; *printf* is a function and not an intrinsic feature of the language. It is a function which you might have written, but it already happens to be provided in the file, *c.lib*. This file is a library of all the standard i/o functions. It also contains many support routines which are called in the code generated by the compiler. These routines aid in integer arithmetic, operating system support, etc.

When the linker sees that a call to *printf* was made, it pulls the function from the library and combines it with the "hello, world" program. The link command would look like this:

```
ln65 hello.r c.lib
```

When *hello.c* was compiled, calls were made to some invisible support functions in the library. So linking without the standard library will cause some unfamiliar symbols to be undefined.

The modules in *c.lib* have been compiled with the native code compiler, *cg65*. You can alternatively link your programs with *ci.lib*, which has the same modules as *c.lib*, except that they have been compiled with *cgi* instead of *cg65*.

The Linking Process

Since the standard library contains only a limited number of general purpose functions, all but the most trivial programs are certain to call user-defined functions. It is up to the linker to connect a function call with the definition of the function somewhere in the code.

In the example given below, the linker will find two function calls in file 1. The reference to *func1* is "resolved" when the definition of *func1* is found in the same file. The following command

```
ln65 file1.r c.lib
```

will cause an error indicating that *func2* is an undefined symbol. The reason is that the definition of *func2* is in another file, namely *file2.r*. The linkage has to include this file in order to be successful:

```
ln65 file1.r file2.r c.lib
```

<i>file 1</i>	<i>file 2</i>
main()	func2()
{	{
func1();	return;
func2();	}
}	
func1()	
{	
return;	
}	

Libraries

A library is a collection of object files put together by a librarian. Libraries intended for use with *ln65* must be built with the Manx librarian, *lb*. This utility is described in the Utility Programs chapter.

All object files specified to the linker will be "pulled into" the linkage; they are automatically included in the final executable file. However, when a library is encountered, it is searched. Only those modules in the library which satisfy a previous function call are pulled in.

For Example

Consider the "hello, world" example. Having looked at the module, *hello.r*, the linker has built a list of undefined symbols. This list includes all the global symbols that have been referenced but not

defined. Global variables and all function names are considered to be global symbols.

The list of undefined symbols for *hello.r* includes the symbol *printf*. When the linker reaches the standard library, this is one of the symbols it will be looking for. It will discover that *printf* is defined in a library module whose name also happens to be *printf* (There is not any necessary relation between the name of a library module and the functions defined within it).

The linker pulls in the *printf* module in order to resolve the reference to the *printf* function.

Files are examined in the order in which they are specified on the command line. So the following linkages are equivalent:

```
ln65 hello.r
```

```
ln65 c.lib hello.r
```

Since no symbols are undefined when the linker searches *c.lib* in the second line, no modules are pulled in. It is good practice to leave all libraries at the end of the command line, with the standard library last of all.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant. The linker searches a library once, from beginning to end. If a module is pulled in at any point, and that module introduces a new undefined symbol, then that symbol is added to the running list of undefined symbols. The linker will not search the library twice to resolve any references which remain unresolved. A common error lies in the following situation:

<i>module of program</i>	<i>references (function calls)</i>
main.r	getinput, do__calc
input.r	gets
calc.r	put__value
output.r	printf

Suppose we build a library to hold the last three modules of this program. Then our link step will look like this:

```
ln65 main.r proglib.lib c.lib
```

But it is important that *proglib.lib* is built in the right order. Let's assume that *main()* calls two functions, *getinput()* and *do__calc()*. *getinput()* is defined in the module *input.r*. It in turn calls the standard library function *gets()*. *do__calc()* is in *calc.r* and calls *put__value()*. *put__value()* is in *output.r* and calls *printf()*.

What happens at link time if *proglib.lib* is built as follows?

```
proglib.lib:          input.r
                      output.r
                      calc.r
```

After *main.r*, the linker has *getinput* and *do_calc* undefined (as well as some other support functions in *c.lib*). Then it begins the search of *proglib.lib*. It looks at the library module, *input*, first. Since that module defines *getinput*, that symbol is taken off the list of undefined's. But *gets* is added to it.

The symbols *do_calc* and *gets* are undefined when the linker examines the module, *output*. Since neither of these symbols is defined there, that module is ignored. In the next module, *calc*, the reference to *do_calc* is resolved but *put_value* is a new undefined symbol.

The linker still has *gets* and *put_value* undefined. It then moves on to *c.lib*, where *gets* is resolved. But the call to *put_value* is never satisfied. The error from the linker will look like this:

Undefined symbol: put_value__

This means that the module defining *put_value* was not pulled into the linkage. The reason, as we saw, was that *put_value* was not an undefined symbol when the *output* module was passed over. This problem would not occur with the library built this way:

```
proglib.lib:          input.r
                      calc.r
                      output.r
```

The standard libraries were put together with much care so that this kind of problem would not arise.

Occasionally it becomes difficult or impossible to build a library so that all references are resolved. In the example, the problem could be solved with the following command:

```
ln65 main.r proglib.lib proglib.lib c.lib
```

The second time through *proglib.lib*, the linker will pull in the module *output*. The reason this is not the most satisfactory solution is that the linker has to search the library twice; this will lengthen the time needed to link.

2. Using the Linker

The general form of a linkage is as follows:

```
ln65 [-options] file1.r [file2.r ...] [lib1.lib ...]
```

The linker combines object modules produced by the *as65* and/or *asi* assemblers into an executable program. It can search libraries of object modules for functions needed to complete the linkage; including just the needed modules in the executable program. The linker makes just a single pass through a library, so that only forward references within a library will be resolved.

The executable file

The name of the executable output file can be selected using the *-O* linker option. If this option isn't used, the linker will derive the name of the output file from that of the first object file listed on the command line, by deleting its extension. In the default case, the executable file will be located in the directory in which the first object file is located. For example,

```
ln65 prog.r c.lib
```

will produce the file *prog*. The standard library, *c.lib*, will have to be included in most linkages.

A different output file can be specified with the *-O* option, as in the following command:

```
ln65 -o program mod1.r mod2.r c.lib
```

This command also shows how several individual modules can be linked together. A "module", in this sense, is a section of a program containing a limited number of functions, usually related. These modules are compiled and assembled separately and linked together to produce an executable file.

Libraries

Function source is provided with CG65, with which you can generate several libraries. Two of these libraries are *c.lib* and *ci.lib*, which contain general-purpose functions. The other two are *m.lib* and *mi.lib*, which contain floating point functions. The modules in *c.lib* and *m.lib* have been compiled with the native code compiler, while those in *ci.lib* and *mi.lib* have been compiled with the pseudo code compiler.

All programs must be linked with one of the versions of *c.lib*. In addition to containing 6502 versions of all the non-floating point functions described in the Functions chapter, it contains internal functions which are called by compiler-generated code, such as functions to perform long arithmetic.

Programs that perform floating point operations must be linked with one of the versions of *m.lib*, in addition to a version of *c.lib*. The

floating point library must be specified on the linker command line before *c.lib*.

You can also create your own object module libraries using the *lb* program. These libraries must be listed on the linker command line before the Manx libraries.

For example, the following links the module *program.r*, searching the libraries *mylib.lib*, *new.lib*, *m.lib*, and *c.lib* for needed modules:

```
ln65 program.r mylib.lib new.lib m.lib c.lib
```

Each of the libraries will be searched once in the order in which they appear on the command line.

Libraries can be conveniently specified using the *-L* option. For example, the following command is equivalent to the following:

```
ln65 -o program.r -lmylib -lnew -lm -lc
```

For more information, see the description of the *-L* option in the Options section of this chapter.

3. Linker Options

3.1 Summary of options

3.1.1 General Purpose Options

- O *file* Write executable code to the file named *file*.
- L*name* Search the library *name.lib* for needed modules.
- F *file* Read command arguments from *file*.
- T Generate a symbol table file.
- M Don't issue warning messages.
- N Don't abort if there are undefined symbols.
- V Be verbose.

3.1.2 Options for Segment Address Specification

- B *addr* Set the program's base address to the hex value *addr*.
- C *addr* Set the starting address of the program's code segment to the hex value *addr*.
- D *addr* Set the starting address of the program's data segment to the hex value *addr*.
- U *addr* Set the starting offset of the program's uninitialized data segment to the hex value *addr*.

3.1.3 Options for Overlay Usage

- R Create a symbol table to be used when linking overlays.
- +C *size* Reserve *size* bytes at end of the program's code segment (the overlay's code segment is loaded here). *size* is a hex value.
- +D *size* Reserve *size* bytes at end of the program's initialized and uninitialized data segments (the overlay's data is loaded here). *size* is a hex value.

3.1.4 65xx Options

- +H *start,end* Define a hole in the program, whose beginning and ending addresses are the hex values *start* and *end*.

3.2 Detailed description of the options

3.2.1 General Purpose Options:

The -O option

The -O option can be used to specify the name of the file to which the linker is to write the executable program. The name of this file is in the parameter that follows the -O. For example, the following command writes the executable program to the file *progout*.

```
ln65 -o progout prog.o c.lib
```

If this option isn't used, the linker derives the name of the executable file from that of the first input file, by deleting its extension.

The -L option

The -L option provides a convenient means of specifying to the linker a library that it should search, when the extension of the library is *.lib*.

The name of the library is derived by concatenating the value of the environment variable *CLIB65*, the letters that immediately follow the -L option, and the string *.lib*. For example, with the libraries *subs.lib*, *io.lib*, *m.lib*, and *c.lib* in a directory specified by *CLIB65*, you can link the module *prog.o*, and have the linker search the libraries for needed modules by entering

```
ln65 prog.o -lsubs -lio -lm -lc
```

The command that sets *CLIB65* varies from system to system. On PC DOS, the *set* command is used. For example, the following command defines *CLIB65* when the libraries are in the directory */cg65/lib*:

```
set CLIB65=/cg65/lib/
```

Note the terminating slash on the *CLIB65* variable: this is required since the linker simply prepends the value of the *CLIB65* variable to the -L string.

The -F option

-F *file* causes the linker to merge the contents of the given file with command line arguments. For example, the following command causes the linker to create an executable program in the file *myprog*. The linker includes the modules *myprog.o*, *mod1.o*, and *mod2.o* in the program, and searches the libraries *mylib.lib* and *c.lib* for needed modules.

```
ln65 myprog.o -f argfil c.lib
```

where the file *argfil*, contains the following:

```
mod1.o mod2.o  
mylib.lib
```

The linker arguments in *argfile* can be separated by tabs, spaces, or newline characters.

There are several uses for the *-F* option. The most obvious is to supply the names of modules that are frequently linked together. Since all the modules named are automatically pulled into the linkage, the linker does not spend any time in searching, as with a library. Furthermore, any linker option except *-F* can be given in a *-F* file. *-F* can appear on the command line more than once, and in any order. The arguments are processed in the order in which they are read, as always.

The *-T* option

The *-T* option causes the linker to write a program's symbol table to a file. You must specify this option if the generated program is going to be converted into Intel hex records by *hex65*.

Each line of the symbol table file contains a symbol name and its address.

The symbol table file will have the same name as that of the file containing the executable program, with extension changed to *.sym*.

There are several special symbols which will appear in the table. They are defined in the Memory Organization section of the Technical Information chapter.

The *-M* option

The linker issues the message "multiply defined symbol" when it finds a symbol that is defined with the assembly language directives *global* or *public* in more than one module. The *-M* option causes the linker to suppress this message unless the symbol is defined in more than one *public* directive.

To maintain compatibility with previous versions of Aztec C, the linker will generate code for a variable that is defined in multiple *global* statements and in at most one *public* statement, and also issue the "multiply defined symbol" message. Thus, if you use the *global* and *public* directives in this way, and don't want to get this message, use the *-M* option to suppress them.

The definition of a symbol in more than one *public* directive is never valid, so the *-M* option doesn't suppress messages in this case.

For more information, see the discussion on global symbols in the Programmer Information sections of the Compiler and Assembler chapters.

The -N option

Normally, the linker halts without generating an executable program if there are undefined symbols; The -N option causes the linker to go ahead and generate an executable program anyway.

The -V option

The -V option causes the linker to send a progress report of the linkage to the screen as each input file is processed. This is useful in tracking down undefined symbols and other errors which may occur while linking.

3.2.2 Options for segment address specification

The linker organizes a program into three segments: code, initialized data, and uninitialized data areas. You can define the starting address of these segments using the -C, -D, and -U linker options, respectively. A fourth linker option, -B, will set the "base address" of the program. These options are followed by the desired offset, in hex.

By default, the base address of a program is 0x800. Also by default, a program's code segment starts three bytes after the base address, its initialized data segment follows the code, and its uninitialized data follows the initialized data.

A file created by the linker begins with a 4-byte header; this is followed by a memory image of the program, from its base address through the end of its code or initialized data segments (whichever is higher). This image can be loaded into memory, with the first byte in the file loaded at the program's base address.

The base address

By default, the linker assumes that a program will begin execution at its base address, and so creates a jump instruction and places it at the program's base address. This jump instruction, when executed, transfers control to the program's startup routine, which is usually somewhere in the middle of the program's code segment. A startup routine performs initialization activities and then calls the program's *main* function.

The linker won't generate the base address jump instruction if there isn't room for it in program's memory image; that is, if the segment (code, initialized data, or uninitialized data) that is closest to the base address begins less than three bytes above the base address.

The startup routine

A program's startup routine is defined using the assembly language *entry* directive. If, among the modules that are linked together into an executable program, the linker finds one that contains the *entry* directive, the location specified in that directive is used as the

program's entry point. If none of the linked modules contain an *entry* directive, the start of the program's code segment is used as the program's entry point.

The presence of an *entry* directive in a library module, however, does not cause the linker to include that module in a program that it's building. Inclusion of a library module in a program is caused only when one of the module's globally-accessible symbols (defined by specifying the symbol in a *public* directive) is also on the linker's list of undefined symbols.

For example, the *rom* startup routine contains the directives *public .begin* and *entry .begin*. By default, the compiler generates a reference to *.begin* when it compiles any module; this reference causes the linker, when it encounters the *rom* module in *c.lib*, to include the *rom* module in the program it's building; the module's *entry .begin* directive then causes the linker to define *.begin* as the program's entry point.

Example 1

In a typical 65xx ROM system, the ROM is at the top of the memory space, and the RAM is at the bottom. The fields in the 65xx memory space between 0xffffa and 0xfffff contain pointers of locations to which the 65xx will transfer control upon the occurrence of special events such as power-up, system reset, and receipt of an interrupt. Hence the code for a 65xx ROM system is usually placed near the top of memory, so that the same ROM can contain both the program's code and the special pointers. Pages 0 and 1, which occupy memory locations 0 through 0xff and 0x100 through 0x1ff, are special on a 65xx, and always contains RAM. Hence the data for a 65xx ROM system is usually placed just above pages 0 and 1, so that the same RAM that is used for these two pages can also hold the program's data.

Since, on a typical ROM system, the two bytes beginning at 0xffffc contain the address to which the processor will transfer control on system reset or power-up, there is no need for the linker's base address jump instruction. So for a typical ROM system, the base address and the beginning of the data segments are set to the same value.

For example, the following command creates the memory image of a program that will be burned into ROM, where its code begins at 0xf000, its initialized data at 0x200, its uninitialized data immediately following the initialized data:

```
ln65 -b 200 -d 200 -c f000 progr -lc
```

Example 2

In some cases, a ROM program fits into another ROM system; a system whose ROM occupies the high section of memory, handling interrupts, power-up, etc, and whose RAM occupies the low section of memory. In this case, the add-on ROM program will fit somewhere in

the middle of the 65xx memory space, with its code beginning at a known place so that separately-linked ROM programs can access it by issuing a call to that known place. If, in this case, the add-on program's code is below its data, use can be made of the linker's generation of a jump instruction at the program's base address to the program's entry point. That is, the program's base address is set to that known address, the beginning of the program's code segment is set three bytes past the base address, and the program's data segments are placed somewhere above the code segment.

For example, the following command links such a program, where its base address begins at 0x8000, its code at 0x8003, its initialized data at 0xa000, and its uninitialized data immediately after the initialized data:

```
ln65 -b 8000 -d a000 prog.r -lc
```

It wasn't necessary to use the `-C` option to explicitly specify the starting address of the code segment; by default, it starts three bytes after the base address.

Example 3

In this example, we want to modify the example 2 program slightly, so that the program's data is below its code. In this case, you can't make use of the linker's automatic generation of a jump instruction to the program's entry point, since this instruction won't be burned into ROM.

For this, you must explicitly specify the module that contains the program's entry point, as the first module to be linked, which causes the linker to place it at the beginning of the program's code segment. And you must explicitly tell the linker that the program's code segment begins at the program's "known address"; that is, the address that other programs will call to access the program. For example, the following command links a program so that its code begins at 0x8000, its initialized data at 0x4000, and its uninitialized data right after the initialized data:

```
ln65 -b 4000 -d 4000 -c 8000 -o prog rom.r prog.r -lc
```

Setting the program's base address equal to the address at which its data begins tells the linker not to generate a jump instruction at the base address. *rom.r* is the startup module, which could have been obtained by extracting it from *c.lib*. *prog.r* contains the main body of the program, including its *main* function.

3.2.3 Options for Overlay Usage

The `-R` option causes the linker to generate a file containing the symbol table. It's used when linking a program which calls overlays.

The name of the symbol table file is derived from that of the executable file by changing the extension to *.rsm*. The file is placed in the same directory as the executable file.

The linker reserves space in a program between its uninitialized data area and its heap, into which the program's overlays will be loaded. The amount of space equals the sum of the values that you define using the *+C* and *+D* options. For example,

```
ln65 +c 3000 +d 1000 prog.o -lc
```

will reserve 0x4000 bytes for overlays. See the Overlay section of the Technical Information chapter for more details.

3.2.4 65xx options

The +H Option

The *+H* option defines a "hole"; that is, an area of memory into which the linker should not place a program's code or data. You can create at most four holes in a program using *+H* options.

The option has the following form:

```
+h start,end
```

where *start* and *end* are the addresses, in hex, of the hole's starting and ending addresses.

For example, suppose you want to create a program, *line*, that begins at address 0x800, and that the program is going to access a graphics area that resides between addresses 0x2000-0x4000. The following command will link the program:

```
ln65 +h 2000,4000 line.o -lc
```

The linker will place as much of the program's code and data as possible in the area between 0x800-0x2000, and place any additional code and data in the area above 0x4000.

The linker creates a program's code segment by concatenating module code segments, until and unless a module's code overlaps a reserved area. If this occurs, the linker moves the module's entire code segment above the reserved area, in the first non-reserved area in which it will entirely fit, and then continues the concatenation of module code segments.

The linker creates a program's initialized data segment in the same way: it concatenates module initialized data segments as much as possible, without overlapping a reserved area and without breaking a module's initialized data segment into discontinuous pieces.

Because the linker won't break up a module's code segment or data segment, it's likely that some space below a hole will be left unused by the linker.

UTILITY PROGRAMS

Chapter Contents

Utility Programs	util
arcv	4
cnm65	5
crc	9
hd	10
hex65	11
lb65	14
make	25
mkarcv	4
obd65	43
optint65	44
ord65	45
sqz65	46

Utility Programs

This chapter describes utility programs that are provided with Aztec CG65.

NAME

arcv & mkarcv - source dcarchiver & archiver

SYNOPSIS

arcv *arcfile* [*destpfix*]
mkarcv *arcfile*

DESCRIPTION

arcv extracts the source from the archive *arcfile*, which has been previously created by *mkarcv*.

destpfix defines the directory in which the generated files are placed: if it is not specified, the generated files are placed in the current directory. If it is specified, it is prepended to the name of the file that *arcv* would otherwise use.

mkarcv creates the archive file *arcfile*, placing in it the files whose names it reads from its standard input. Only one file name is read from a standard input line.

EXAMPLES

For example, the file *header.arc* contains the source for all the header files. To create these header files in the current directory, enter:

```
arcv header.arc
```

The following command creates the archive *myarc.arc* containing the files *in.c*, *out.c*, and *hello.c*.

```
mkarcv myarc.arc <myarc.bld
```

The names of the following three files are contained in the file *myarc.bld*:

```
in.c  
out.c  
hello.c
```

NAME

cnm65 - display object file info

SYNOPSIS

cnm65 [-sol] file [file ...]

DESCRIPTION

cnm65 displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler, libraries of object modules created by the *lb* librarian, and, when applicable, 'rsm' files created by the Manx linker during the linking of an overlay root.

For example, the following displays the size and symbols for the object module *sub1.o* and the library *c.lib*:

```
cnm65 sub1.o c.lib
```

By default, the information is sent to the console. It can be redirected to a file or device in the normal way. For example, the following commands send information about *sub1.o* to the display and to the file *dispfile*:

```
cnm65 sub1.o
cnm65 sub1.o > dispfile
```

The first line listed by *cnm65* for an object module has the following format:

```
file (module): code: cc  data: dd  udata: uu  total: tt (0xhh)
```

where

- * *file* is the name of the file containing the module,
- * *module* is the name of the module; if the module is unnamed, this field and its surrounding parentheses aren't printed;
- * *cc* is the number of bytes in the module's code segment, in decimal;
- * *dd* is the number of bytes in the module's initialized data segment, in decimal;
- * *uu* is the number of bytes in the module's uninitialized data segment, in decimal;
- * *tt* is the total number of bytes in the module's three segments, in decimal;
- * *hh* is the total number of bytes in the module's three segments, in hexadecimal.

If *cnm65* displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the modules' code segments, initialized data segments, and uninitialized data segments, and the sum of the sizes of all segments of all modules. Each sum is in decimal; the total of all segments is also

given in hexadecimal.

The *-s* option tells *cnm65* to display just the sizes of the object modules. If this option isn't specified, *cnm65* also displays information about each named symbol in the object modules.

When *cnm65* displays information about the modules' named symbols, the *-l* option tells *cnm65* to display each symbol's information on a separate line and to display all of the characters in a symbol's name; if this option isn't used, *cnm65* displays the information about several symbols on a line and only displays the first eight characters of a symbol's name.

The *-o* option tells *cnm65* to prefix each line generated for an object module with the name of the file containing the module and the module name in parentheses (if the module is named). If this option isn't specified, this information is listed just once for each module: prefixed to the first line generated for the module.

The *-o* option is useful when using *cnm65* in combination with *grep*. For example, the following commands will display all information about the module *perror* in the library *c.lib*:

```
cnm65 -o c.lib >tmp
grep perror tmp
```

cnm65 displays information about an module's 'named' symbols; that is, about the symbols that begin with something other than a dollar sign followed by a digit. For example, the symbol *quad* is named, so information about it would be displayed; the symbol *\$0123* is unnamed, so information about it would not be displayed.

For each named symbol in a module, *cnm65* displays its name, a two-character code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

If the first character of a symbol's type code is lower case, the symbol can only be accessed by the module; that is, it's local to the module. If this character is upper case, the symbol is global to the module: either the module has defined the symbol and is allowing other modules to access it or the module needs to access the symbol, which must be defined as a global or public symbol in another module. The type codes are:

ab The symbol was defined using the assembler's EQU directive. The value listed is the equated value of its symbol.

 The compiler doesn't generate symbols of this type.

pg The symbol is in the code segment. The value is the offset of the symbol within the code segment.

- The compiler generates this type symbol for function names. Static functions are local to the function, and so have type *pg*; all other functions are global, that is, callable from other programs, and hence have type *Pg*.
- dt* The symbol is in the initialized data segment. The value is the offset of the symbol from the start of the data segment.
- The compiler generates symbols of this type for initialized variables which are declared outside any function. Static variables are local to the program and so have type *dt*; all other variables are global, that is, accessible from other programs, and hence have type *Dt*.
- ov* When an overlay is being linked and that overlay itself calls another overlay, this type of symbol can appear in the rsm file for the overlay that is being linked. It indicates that the symbol is defined in the program that is going to call the overlay that is being linked.
- The value is the offset of the symbol from the beginning of the physical segment that contains it.
- un* The symbol is used but not defined within the program. The value has no meaning.
- In assembly language terms, a type of *Un* (the U is capitalized) indicates that the symbol is the operand of a *public* directive and that it is perhaps referenced in the operand field of some statements, but that the program didn't create the symbol in a statement's label field.
- The compiler generates *Un* symbols for functions that are called but not defined within the program, for variables that are declared to be *extern* and that are actually used within the program, and for uninitialized, global dimensionless arrays. Variables which are declared to be *extern* but which are not used within the program aren't mentioned in the assembly language source file generated by the compiler and hence don't appear in the object file.
- bs* The symbol is in the uninitIALIZED data segment. The value is the space reserved for the symbol.
- The compiler generates *bs* symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *bs* symbols for symbols defined using the *bss* assembler directive.

Gl

The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates *Gl* symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *Gl* symbols for variables declared using the *global* directive which have a non-zero size.

NAME

`crc` - Utility for generating the CRC for files

SYNOPSIS

`crc file1 file2 ...`

DESCRIPTION

crc computes a number, called the CRC, for the specified *files*.

The CRC for a file is entirely dependent on the file's contents, and it is very unlikely that two files whose contents are different will have the same CRCs. Thus, *crc* can be used to determine whether a file has the expected contents.

As an example of the usage of *crc*, the following command computes the crc of all files whose extension is *.c*:

`crc *.c`

NAME

hd - hex dump utility

SYNOPSIS

hd [+n[.]] file1 [+n[.]] file 2 ...

DESCRIPTION

hd displays the contents of one or more files in hex and ascii to its standard output.

file1, *file2*, ... are the names of the files to be displayed.

+n specifies the offset into the file where the display is to start, and defaults to the beginning of the file. If *+n* is followed by a period, *n* is assumed to be a decimal number; otherwise, it's assumed to be hexadecimal. Each file will be displayed beginning at the last specified offset.

EXAMPLES

The following command displays the contents of files *oldtest* and *newtest*, beginning at offset 0x16b, and of the file named *junk*, beginning at its first byte:

hd +16b oldtest newtest +0 junk

The next command displays the contents of *tstfil*, beginning at byte 1000:

hd -r +1000. tstfil

NAME

hex65 - Intel hex generator

SYNOPSIS

hex65 [-options] progfile

DESCRIPTION

hex65 translates a program that was generated by the Aztec CG65 linker, into Intel hex records. The program can then be burned into ROM by feeding the hex records into a ROM programmer. The records are written to one or more files, each of which contains the hex records for one ROM chip.

The ROM chips that are generated from the *hex65* output files will contain the program's code, followed by a copy of its initialized data.

Note: when a ROM system is started, its RAM contains random values; the Aztec CG65 startup routine sets up its initialized data area, using the copy that's in ROM.

Optionally, the last ROM chip will occupy the top section of the 65xx memory space, and contain in the top 6 bytes, pointers to the program's power-up/reset routine, the nmi interrupt handler, and the irq interrupt handler.

hex65 assumes that the size of each ROM chip is 2 kb. You can explicitly define the size of each ROM using *hex65*'s *-P* option.

The input files

When you tell the linker to create the memory image of a program that's to be burned into ROM, you must specify the *-T* option, to make the linker also create a file containing the program's symbol table. That's because when *hex65* translates the memory image of a program into hex records, it reads both of these files.

The names of the files that are read by *hex65* must obey the linker's conventions: the memory image file should not have an extension, and the name of the symbol table file should be the same as that of the memory image file, with extension *.sym*.

The only file name you specify when you start *hex65* is that of the memory image file; *hex65* derives the name of the symbol table file by appending *.sym* to it.

The output files

hex65 derives the name of each output file from that of the file that contains the memory image, by appending an extension of the form *.xnn*, where *nn* is a number. For example, if the name of the memory image file is *prog*, then the name of the output files generated by *hex65* are *prog.x00*, *prog.x01*, and so on, where the *.x00* file

contains the hex records for the lowest-addressed ROM, *.x01* the hex records for the next ROM, etc. When *hex65* generates hex records that will initialize the 65xx power-up and interrupt vector fields, it will create a separate file, if necessary, that contains just these Intel hex records. The extension of this separate file indicates the position of its ROM in the memory space.

For example, suppose that *hex65* is creating hex records for a program whose code and copy of initialized data will reside in two 2-kb ROMs that begin at 0xe000, and that it is also generating the hex records that will initialize the power-up and interrupt vectors. Then *hex65* will create the following files, of which the first two contain the records for the code and copy of initialized data and the third the records for the vectors:

- prog.x00* Contains the hex records for the ROM chip that occupies 0xe000-0xe7ff;
- prog.x01* Contains the hex records for the ROM that occupies 0xe800-0xefff;
- prog.x03* Contains the hex records for the ROM that occupies 0xf800-0xffff.

The position of each file's corresponding ROM in the memory space is indicated by the number in its file's extension:

- * The number in the first file's extension is 00, so its ROM occupies the 2-kb block that begins at 0xe000+0*0x400. Note: nothing in the names of these files indicates the memory location of these ROMs, but you know that the first one begins at the starting address of the program's code segment; that is, at 0xe000.
- * The number in the second file's extension is 01, so its ROM occupies the 2-kb block that begins at 0xe000+1*0x400.
- * The number in the third file's extension is 03, so its ROM occupies the 2-kb block that begins at 0xe000+2*0x400.

The options

hex65 supports the following options:

- Pnn* Each ROM is *nn* bytes long, where *nn* is a decimal number. If this option isn't specified, each ROM is assumed to be 2 kb long.
- Z* Don't generate hex records for the power-up and interrupt vectors. If this option isn't specified, these vectors are generated.
- Bnnnn* The program's base address is 0xn timer (this is the address that was specified as the base address when the program was linked, using either the *-B* option or the

default value). If this option isn't specified, it's assumed to be the lesser of the beginning addresses of the program's code or initialized data segments.

- S Output spaces between the fields of each hex record, to make the records more readable.
- L Output hex digits using lower case characters.
- ? List the options.

NAME

lb65 - object file librarian

SYNOPSIS

lb65 library [options] [mod1 mod2 ...]

DESCRIPTION

lb65 is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.

This description of *lb65* is divided into three sections: the first describes briefly *lb65*'s arguments and options, the second *lb65*'s basic features, and the third the rest of *lb65*'s features.

1. The arguments to *lb65*

1.1 The *library* argument

When started, *lb65* acts upon a single library file. The first argument to *lb65* (*library*, in the synopsis) is the name of this file. The filename extension for *library* is optional; if not specified, it's assumed to be *.lib*.

1.2 The *options* argument

There are two types of *options* argument: function code options, and qualifier options. These options will be summarized in the following paragraphs, and then described in detail below.

1.2.1 Function code options

When *lb65* is started, it performs one function on the specified library, as defined by the *options* argument. The functions that *lb65* can perform, and their corresponding option codes, are:

<i>function</i>	<i>code</i>
create a library	(no code)
add modules to a library	-a, -i, -b
list library modules	-t
move modules within a library	-m
replace modules	-r
delete modules	-d
extract modules	-x
ensure module uniqueness	-u
define module extension	-e
help	-h

In the synopsis, the *options* argument is surrounded by square brackets. This indicates that the argument is optional; if a code isn't specified, *lb65* assumes that a library is to be created.

1.2.2 Qualifier options

In addition to a function code, the *options* argument can optionally specify a qualifier, that modifies *lb65*'s behavior as it is performing the requested function. The qualifiers and their codes are:

verbose	-v
silent	-s

The qualifier can be included in the same argument as the function code, or as a separate argument. For example, to cause *lb65* to append modules to a library, and be silent when doing it, any of the following option arguments could be specified:

- as
- sa
- a -s
- s -a

1.3 The *mod* arguments

The arguments *mod1*, *mod2*, etc. are the names of the object modules, or the files containing these modules, that *lb65* is to use. For some functions, *lb65* requires an object module name, and for others it requires the name of a file containing an object module. In the latter case, the file's extension is optional; if not specified, the *lb65* that's supplied with native Aztec C systems assumes that it's *.o*, and the *lb65* that's supplied with cross development versions of Aztec C assumes that the extension is *.r*. You can explicitly define the default module extension using the *-e* option.

1.4 Reading arguments from another file

lb65 has a special argument, *-f filename*, that causes it to read command line arguments from the specified file. When done, it continues reading arguments from the command line. Arguments can be read from more than one file, but the file specified in a *-f filename* argument can't itself contain a *-f filename* argument.

2. Basic features of *lb65*

In this section we want to describe the basic features of *lb65*. With this knowledge in hand, you can start using *lb65*, and then read about the rest of the features of *lb65* at your leisure.

The basic things you need to know about *lb65*, and which thus are described in this section, are:

- * How to create a library
- * How to list the names of modules in a library
- * How modules get their names

- * Order of modules in a library
- * Getting *lb65* arguments from a file

Thus, with the information presented in this section you can create libraries and get a list of the modules in libraries. The third section of this description shows you how to modify selected modules within a library.

2.1 Creating a Library

A library is created by starting *lb65* with a command line that specifies the name of the library file to be created and the names of the files whose object modules are to be copied into the library. It doesn't contain a function code, and it's this absence of a function code that tells *lb65* that it is to create a library.

For example, the following command creates the library *exmpl.lib*, copying into it the object modules that are in the files *obj1.o* and *obj2.o*:

```
lb65 exmpl.lib obj1.o obj2.o
```

Making use of *lb65*'s assumptions about file names for which no extension is specified, the following command is equivalent to the above command:

```
lb65 exmpl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by *lb65*. In this case, all the modules in the input library are copied into the new library.

2.1.1 The temporary library

When *lb65* creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, *lb65* erases the file having the same name as the specified library, and then renames the new library, giving it the name of the specified library. Thus, *lb65* makes sure it can create a library before erasing an existing one.

Note that there must be room on the disk for both the old library and the new.

2.2 Getting the table of contents for a library

To list the names of the modules in a library, use *lb65*'s *-t* option. For example, the following command lists the modules that are in *exmpl.lib*:

```
lb65 exmpl -t
```

The list will include some ****DIR**** entries. These identify blocks within the library that contain control information. They are created and deleted automatically as needed, and cannot be changed by you.

2.3 How modules get their names

When a module is copied into a library from a file containing a single object module (that is, from an object module generated by the Manx assembler), the name of the module within the library is derived from the name of the input file by deleting the input file's volume, path, and extension components.

For example, in the example given above, the names of the object modules in *exmpl.lib* are *obj1* and *obj2*.

An input file can itself be a library. In this case, a module's name in the new library is the same as its name in the input library.

2.4 Order in a library

The order of modules in a library is important, since the linker makes only a single pass through a library when it is searching for modules. For a discussion of this, see the tutorial section of the Linker chapter.

When *lb65* creates a library, it places modules in the library in the order in which it reads them. Thus, in the example given above, the modules will be in the library in the following order:

obj1 obj2

As another example, suppose that the library *oldlib.lib* contains the following modules, in the order specified:

sub1 sub2 sub3

If the library *newlib.lib* is created with the command

lb65 newlib mod1 oldlib.lib mod2 mod3

the contents of the newly-created *newlib.lib* will be:

mod1 sub1 sub2 sub3 mod2 mod3

The *ord* utility program can be used to create a library whose modules are optimally sorted. For information, see its description later in this chapter.

2.5 Getting *lb65* arguments from a file

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to *lb65* on a single command line. In this case, *lb65*'s *-f filename* feature can be of use: when *lb65* finds this option, it opens the specified file and starts reading command arguments from it. After finishing the file, it continues to scan the command line.

For example, suppose the file *build* contains the line

exmpl obj1 obj2

Then entering the command

```
lb65 -f build
```

causes *lb65* to get its arguments from the file *build*, which causes *lb65* to create the library *exmpl.lib* containing *obj1* and *obj2*.

Arguments in a *-f* file can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a *-f* file can be on separate lines, if desired.

The *lb65* command line can contain multiple *-f* arguments, allowing *lb65* arguments to be read from several files. For example, if some of the object modules that are to be placed in *exmpl.lib* are defined in *arith.inc*, *input.inc*, and *output.inc*, then the following command could be used to create *exmpl.lib*:

```
lb65 exmpl -f arith.inc -f inputinc -f outputinc
```

A *-f* file can contain any valid *lb65* argument, except for another *-f*. That is, *-f* files can't be nested.

3. Advanced *lb65* features

In this section we describe the rest of the functions that *lb65* can perform. These primarily involve manipulating selected modules within a library.

3.1 Adding modules to a library

lb65 allows you to add modules to an existing library. The modules can be added before or after a specified module in the library or can be added to the beginning or end of the library.

The options that select *lb65*'s add function are:

<i>option</i>	<i>function</i>
<i>-b target</i>	add modules before the module <i>target</i>
<i>-i target</i>	same as <i>-b target</i>
<i>-a target</i>	add modules after the module <i>target</i>
<i>-b+</i>	add modules to the beginning of the library
<i>-i+</i>	same as <i>-b+</i>
<i>-a+</i>	add modules to the end of the library

In an *lb65* command that selects the *add* function, the names of the files containing modules to be added follows the add option code (and the target module name, when appropriate). A file can contain a single module or a library of modules.

Modules are added in the order that they are specified. If a library is to be added, its modules are added in the order they occur in the input library.

3.1.1 Adding modules before an existing module

As an example of the addition of modules before a selected module, suppose that the library *exmpl.lib* contains the modules

```
obj1  obj2  obj3
```

The command

```
lb65 exmpl -i obj2 mod1 mod2
```

adds the modules in the files *mod1.o* and *mod2.o* to *exmpl.lib*, placing them before the module *obj2*. The resultant *exmpl.lib* looking like this:

```
obj1  mod1  mod2  obj2  obj3
```

Note that in the *lb65* command we didn't need to specify the extension of either the file containing the library to which modules were to be added or the extension of the files containing the modules to be added. *lb65* assumed that the extension of the file containing the target library was *.lib*, and that the extension of the other files was *.o*.

As an example of the addition of one library to another, suppose that the library *mylib.lib* contains the modules

```
mod1  mod2  mod3
```

and that the library *exmpl.lib* contains

```
obj1  obj2  obj3
```

Then the command

```
lb65 -b obj2 mylib.lib
```

adds the modules in *mylib.lib* to *exmpl.lib*, resulting in *exmpl.lib* containing

```
obj1  mod1  mod2  mod3  obj2  obj3
```

Note that in this example, we had to specify the extension of the input file *mylib.lib*. If we hadn't included it, *lb65* would have assumed that the file was named *mylib.o*.

3.1.2 Adding modules after an existing module

As an example of adding modules after a specified module, the command

```
lb65 exmpl -a obj1 mod1 mod2
```

will insert *mod1* and *mod2* after *obj1* in the library *exmpl.lib*. If *exmpl.lib* originally contained

```
obj1  obj2  obj3
```

then after the addition, it contains

```
obj1  mod1  mod2  obj2  obj3
```

3.1.3 Adding modules at the beginning or end of a library

The options *-b+* and *-a+* tell *lb65* to add the modules whose names follow the option to the beginning or end of a library, respectively. Unlike the *-i* and *-a* options, these options aren't followed by the name of an existing module in the library.

For example, given the library *exmpl.lib* containing

```
obj1  obj2
```

the following command will add the modules *mod1* and *mod2* to the beginning of *exmpl.lib*:

```
lb65 exmpl -i+ mod1 mod2
```

resulting in *exmpl.lib* containing

```
mod1 mod2 obj1 obj2
```

The following command will add the same modules to the end of the library:

```
lb65 exmpl -a+ mod1 mod2
```

resulting in *exmpl.lib* containing

```
obj1  obj2  mod1  mod2
```

3.2 Moving modules within a library

Modules which already exist in a library can be easily moved about, using the *move* option, *-m*.

As with the options for adding modules to an existing library, there are several forms of *move* functions:

<i>option</i>	<i>meaning</i>
<i>-mb target</i>	move modules before the module <i>target</i>
<i>-ma target</i>	move modules after the module <i>target</i>
<i>-mb+</i>	move modules to the beginning of the library
<i>-ma+</i>	move modules to the end of the library

In the *lb65* command, the names of the modules to be moved follows the 'move' option code.

The modules are moved in the order in which they are found in the original library, not in the order in which they are listed in the *lb65* command.

3.2.1 Moving modules before an existing module

As an example of the movement of modules to a position before an existing module in a library, suppose that the library *exmpl.lib* contains

obj1 obj2 obj3 obj4 obj5 obj6

The following command moves *obj3* before *obj2*:

```
lb65 exmpl -mb obj2 obj3
```

putting the modules in the order:

obj1 obj3 obj2 obj4 obj5 obj6

And, given the library in the original order again, the following command moves *obj6*, *obj2*, and *obj1* before *obj3*:

```
lb65 exmpl -mb obj3 obj6 obj2 obj1
```

putting the library in the order:

obj1 obj2 obj6 obj3 obj4 obj5

As an example of the movement of modules to a position after an existing module, suppose that the library *exmpl.lib* is back in its original order. Then the command

```
lb65 exmpl -ma obj4 obj3 obj2
```

moves *obj3* and *obj2* after *obj4*, resulting in the library

obj1 obj4 obj2 obj3 obj5 obj6

3.2.2 Moving modules to the beginning or end of a library

The options for moving modules to the beginning or end of a library are *-mb+* and *-ma+*, respectively.

For example, given the library *exmpl.lib* with contents

obj1 obj2 obj3 obj4 obj5 obj6

the following command will move *obj3* and *obj5* to the beginning of the library:

```
lb65 exmpl -mb+ obj5 obj3
```

resulting in *exmpl.lib* having the order

obj3 obj5 obj1 obj2 obj4 obj6

And the following command will move *obj2* to the end of the library:

```
lb65 exmpl -ma+ obj2
```

3.3 Deleting Modules

Modules can be deleted from a library using *lb65*'s *-d* option. The command for deletion has the form

```
lb65 libname -d mod1 mod2 ...
```

where *mod1*, *mod2*, ... are the names of the modules to be deleted.

For example, suppose that *exmpl.lib* contains

```
obj1  obj2  obj3  obj4  obj5  obj6
```

The following command deletes *obj3* and *obj5* from this library:

```
lb65 exmpl -d obj3 obj5
```

3.4 Replacing Modules

The *lb65* option 'replace' is used to replace one module in a library with one or more other modules.

The 'replace' option has the form *-r target*, where *target* is the name of the module being replaced. In a command that uses the 'replace' option, the names of the files whose modules are to replace the target module follow the 'replace' option and its associated target module. Such a file can contain a single module or a library of modules.

Thus, an *lb65* command to replace a module has the form:

```
lb65 library -r target mod1 mod2 ...
```

For example, suppose that the library *exmpl.lib* looks like this:

```
obj1 obj2 obj3 obj4
```

Then to replace *obj3* with the modules in the files *mod1.o* and *mod2.o*, the following command could be used:

```
lb65 exmpl -r obj3 mod1 mod2
```

resulting in *exmpl.lib* containing

```
obj1  obj2  mod1  mod2  obj4
```

3.5 Uniqueness

lb65 allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

The option *-u* causes *lb65* to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, the *-u* option causes *lb65* to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, suppose that the library *exmpl.lib* contains the following:

```
obj1  obj2  obj3  obj1  obj3
```

The command

```
lb65 exmpl -u
```

will delete the second copies of the modules *obj1* and *obj2*, leaving the library looking like this:

obj1 obj2 obj3

3.6 Extracting modules from a Library

The *lb65* option *-x* extracts modules from a library and puts them in separate files, without modifying the library.

The names of the modules to be extracted follows the *-x* option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it's written to a new file; the file has same name as the module and extension *.o*.

For example, given the library *exmpl.lib* containing the modules

obj1 obj2 obj3

The command

lb65 exmpl -x

extracts all modules from the library, writing *obj1* to *obj1.o*, *obj2* to *obj2.o*, and *obj3* to *obj3.o*.

And the command

lb65 exmpl -x obj2

extracts just *obj2* from the library.

3.7 The 'verbose' option

The 'verbose' option, *-v*, causes *lb65* to be verbose; that is, to tell you what it's doing.

This option can be specified as part of another option, or all by itself. For example, the following command creates a library in a chatty manner:

lb65 exmpl -v mod1 mod2 mod3

And the following equivalent commands cause *lb65* to remove some modules and to be verbose:

lb65 exmpl -dv mod1 mod2

lb65 exmpl -d -v mod1 mod2

3.8 The 'silence' option

The 'silence' option, *-s*, tells *lb65* not to display its signon message.

This option is especially useful when redirecting the output of a list command to a disk file, as described below.

3.9 Rebuilding a library

The following commands provide a convenient way to rebuild a library:

```
lb65 exmpl -st > tfil  
lb65 exmpl -f tfil
```

The first command writes the names of the modules in *exmpl.lib* to the file *tfil*. The second command then rebuilds the library, using as arguments the listing generated by the first command.

The *-s* option to the first command prevents *lb65* from sending information to *tfil* that would foul up the second command. The names sent to *tfil* include entries for the directory blocks, ****DIR****, but these are ignored by *lb65*.

3.10 Defining the default module extension.

Specification of the extension of an object module file is optional; the *lb65* that comes with native development versions of Aztec C assumes that the extension is *.o*, and the *lb65* that comes with cross development versions of Aztec C assumes that it's *.r*. You can explicitly define the default extension using the *-e* option. This option has the form

```
-e .ext
```

For example, the following command creates a library; the extension of the input object module files is *.i*.

```
lb65 my.lib -e .i mod1 mod2 mod3
```

3.11 Help

The *-h* option is provided for brief lapses of memory, and will generate a summary of *lb65* functions and options.

NAME

make - Program maintenance utility

SYNOPSIS

make [-n] [-f makefile] [-a] [name1 name2 ...]

DESCRIPTION

make is a program, similar to the UNIX program of the same name, whose primary function is to create, and keep up-to-date, files that are created from other files, such as programs, libraries, and archives.

When told to make a file, *make* first ensures that the files from which the target file is created are up-to-date or current, recreating just the ones that aren't. Then, if the target file is not current, *make* creates it.

Inter-file dependencies and the commands which must be executed to create files are specified in a file called the 'makefile', which you must write.

make has a rule-processing capability, which allows it to infer, without being explicitly told, the files on which a file depends and the commands which must be executed to create a file. Some rules are built into *make*; you can define others within the makefile.

A rule tells *make* something like this:

"a target file having extension '.x' depends on the file having the same basic name and extension '.y'. To create such a target file, apply the commands ...".

Rules simplify the task of writing a makefile: a file's dependency information and command sequences need be explicitly specified in a makefile only if this information can't be inferred by the application of a rule.

make has a macro capability. A character string can be associated with a macro name; when the macro name is invoked in the makefile, it's replaced by its string.

Preview

The rest of this description of *make* is divided into the following sections:

1. The basics
2. Advanced features
3. Examples

1. The basics

In this section we want to present the basic features of *make*, with which you'll be able to start using *make*. Section 2 describes the other

features of *make*.

Before you can begin using *make*, you must know what *make* does, how to create a simple makefile that contains dependency entries, how to take advantage of *make*'s rule-processing capability, and, finally, how to tell *make* to make a file. Each of these topics is discussed in the following paragraphs.

1.1 What *make* does

The main function of *make* is to make a target file "current", where a file is considered "current" if the files on which it depends are current and if it was modified more recently than its prerequisite files. To make a file current, *make* makes the prerequisite files current; then, if the target file is not current, *make* executes the commands associated with the file, which usually recreates the file.

As you can see, *make* is inherently recursive: making a file current involves making each of its prerequisite files current; making these files current involves making each of their prerequisite files current; and so on.

make is very efficient: it only creates or recreates files that aren't current. If a file on which a target file depends is current, *make* leaves it alone. If the target file itself is current, *make* will announce the fact and halt without modifying the target.

It is important to have the time and date set for *make* to behave properly, since *make* uses the 'last modified' times that are recorded in files' directory entries to decide if a target file is not current.

1.2 The makefile

When *make* starts, one of the first things it does is to read a file, which you must write, called the 'makefile'. This file contains dependency entries defining inter-file dependencies and the commands that must be executed to make a file current. It also contains rule definitions and macro definitions.

In the following paragraphs, we want to just describe dependency entries. In section 2 we discuss the somewhat more advanced topics of rule and macro definition.

A dependency entry in a makefile defines one or more target files, the files on which the targets depend, and the operating system commands that are to be executed when any of the targets is not current. The first line of the entry specifies the target files and the files on which they depend; the line begins with the target file names, followed by a colon, followed by one or more spaces or tabs, followed by the names of the prerequisite files. It's important to place spaces or tabs after the colon that separates target and dependent files; on systems that allow colons in file names, this allows *make* to distinguish

between the two uses of the colon character.

The commands are on the following lines of the dependency information entry. The first character of a command line must be a tab; *make* assumes that the command lines end with the last line not beginning with a tab.

For example, consider the following dependency entry:

```
prog.com: prog.o sub1.o sub2.o
        ln -o prog.com prog.o sub1.o sub2.o -lc
```

This entry says that the file *prog.com* depends on the files *prog.o*, *sub1.o*, and *sub2.o*. It also says that if *prog.com* is not current, *make* should execute the *ln* command. *make* considers *prog.com* to be current if it exists and if it has been modified more recently than *prog.o*, *sub1.o*, and *sub2.o*.

The above entry describes only the dependence of *prog.com* on *prog.o*, *sub1.o*, and *sub2.o*. It doesn't define the files on which the '.o' files depend. For that, we need either additional dependency entries in the makefile or a rule that can be applied to create '.o' files from '.c' files.

For now, we'll add dependency entries in the makefile for *prog.o*, *sub1.o*, and *sub2.o*, which will define the files on which the object modules depend and the commands to be executed when an object module is not current. In section 1.3 we'll then modify the makefile to make use of *make*'s built-in rule for creating a '.o' file from a '.c' file.

Suppose that the '.o' files are created from the C source files *prog.c*, *sub1.c*, and *sub2.c*; that *sub1.c* and *sub2.c* contain a statement to include the file *defs.h* and that *prog.c* doesn't contain any *#include* statements. Then the following long-winded makefile could be used to explicitly define all the information needed to make *prog.com*

```
prog.com: prog.o sub1.o sub2.o
        ln -o prog.com prog.o sub1.o sub2.o -lc

prog.o: prog.c
        cc prog.c

sub1.o: sub1.c defs.h
        cc sub1.c

sub2.o: sub2.c defs.h
        cc sub2.c
```

This makefile contains four dependency entries: for *prog.com*, *prog.o*, *sub1.o*, and *sub2.o*. Each entry defines the files on which its target file depends and the commands to be executed when its target isn't current. The order of the dependency entries in the makefile is not important.

We can use this makefile to make any of the four target files defined in it. If none of the target files exists, then entering

```
make prog.com
```

will cause *make* to compile and assemble all three object modules from their C source files, and then create *prog.com* by linking the object modules together.

Suppose that you create *prog.com* and then modify *sub1.c*. Then telling *make* to make *prog.com* will cause *make* to compile and assemble just *sub1.c*, and then recreate *prog.com*.

If you then modify *defs.h*, and then tell *make* to make *prog.com*, *make* will compile and assemble *sub1.c* and *sub2.c*, and then recreate *prog.com*.

You can tell *make* to make any file defined as a target in a dependency entry. Thus, if you want to make *sub2.o* current, you could enter

```
make sub2.o
```

A makefile can contain dependency entries for unrelated files. For example, the following dependency entries can be added to the above makefile:

```
hello.exe: hello.o
           ln hello.o -lc
hello.o: hello.c
          cc hello.c
```

With these dependency entries, you can tell *make* to make *hello.exe* and *hello.o*, in addition to *prog.com* and its object files.

1.3 Rules

You can see that the makefile describing a program built from many .o files would be huge if it had to explicitly state that each .o file depends on its .c source file and is made current by compiling its source file.

This is where rules are useful. When a rule can be applied to a file that *make* has been told to make or that is a direct or indirect prerequisite of it, the rule allows *make* to infer, without being explicitly told, the name of a file on which the target file depends and/or the commands that must be executed to make it current. This in turn allows makefiles to be very compact, just specifying information that *make* can't infer by the application of a rule.

Some rules are built into *make*; you can define others in a makefile. In the rest of this section, we're going to describe the properties of rules and how you write makefiles that make use of *make*'s built-in rule for creating a .o file from a .c file. For more information on rules,

including a complete list of built-in rules and how to define rules in a makefile, see section 2.2.

1.3.1 *make's* use of rules

A rule specifies a target extension, source extension, and sequence of commands. Given a file that *make* wants to make, it searches the rules known to it for one that meets the following conditions:

- * The rule's target extension is the same as the file's extension;
- * A file exists that has the same basic name as the file *make* is working on and that has the rule's source extension.

If a rule is found that meets these conditions, *make* applies the first such rule to the file it's working on, as follows:

- * The file having the source extension is defined to be a prerequisite of the file with the target extension;
- * If the file having the target extension doesn't have a command sequence associated with it, the rule's commands are defined to be the ones that will make the file current.

One rule built into *make*, for converting *.c* files into *.o* files, says

"a file having extension '*.o*' depends on the file having the same basic name, with extension '*.c*'. To make current such a *.o* file, execute the command

cc x.c

where '*x*' is the name of the file"

Another built-in rule exists for converting *.asm* files into *.o* files, using the Manx assembler.

1.3.2 An example

The *.c* to *.o* rule allows us to abbreviate the long-winded makefile given in section 1.2 as follows:

```
prog.o: prog.o sub1.o sub2.o
    ln -o prog.o prog.o sub1.o sub2.o -lc
sub1.o sub2.o: defs.h
```

In this abbreviated makefile, a dependency entry for *prog.o* isn't needed; using the built-in '*.c* to *.o*' rule, *make* infers that the *prog.o* depends on *prog.c* and that the command *cc prog.c* will make *prog.o* current.

The abbreviated makefile says that both *sub1.o* and *sub2.o* depend on *defs.h*. It doesn't say that they also depend on *sub1.c* and *sub2.c*, respectively, or that the compiler must be run to make them current; *make* infers this information from the *.c* to *.o* rule. The only information given in the dependency entry is that which *make* couldn't

infer by itself: that the two object files depend on *defs.h*.

1.3.3 Interaction of rules and dependency entries

As we showed in the above example, a rule allows you to leave some dependency information unspecified in a makefile. The *prog.o* entry in the long-winded makefile of section 1.2 was not needed, since its information could be inferred by the *.c* to *.o* rule. And the dependence of *sub1.o* and *sub2.o* on their respective C source files, and the commands needed to create the object files was also not needed, since the information could be inferred from the *.c* to *.o* rule.

There are occasions when you don't want a rule to be applied; in this case, information specified in a dependency entry will override that which would be inferred from a rule. For example, the following dependency entry in a makefile

```
add.o:
cc -DFLOAT add.c
```

will cause *add.o* to be compiled using the specified command rather than the command specified by the *.c* to *.o* rule. *make* still infers the dependence of *add.o* on *add.c*, using the *.c* to *.o* rule, however.

2. Advanced features

In the last section we presented the basic features of *make*, with which you can start using *make*. In this section, we present the rest of *make*'s features.

2.1 Dependent Files

A dependent file can be in a different volume or directory than its target file, with the following provisos.

If the file name contains a colon (for example, because the file name defines the volume on which the file is located), the colon must be followed by characters other than spaces or tabs, so that *make* can distinguish between this use of the colon character and its use as a separator between the target and dependent files in a dependency line. This shouldn't be a problem, since most systems don't allow file names to contain spaces or tabs.

All references to a file must use the same name. For example, if a file is referred to in one place using the name

```
/root/src/foo.c
```

then all references to the file must use this exact same name.

On PC DOS and MSDOS, note that the following names may refer to different files:

```
a:dir/sub/foo.c
a:/dir/sub/foo.c.
```

For the first name, the search for *foo.c* begins with the current directory on the *a:* drive; for the second, the search begins with the root directory on the *a:* drive.

2.2 Macros

make has a simple macro capability that allows character strings to be associated with a macro name and to be represented in the makefile by the name. In the following paragraphs, we're first going to describe how to use macros within a makefile, then how they are defined, and finally some special features of macros.

2.2.1 Using macros

Within a makefile, a macro is invoked by preceding its name with a dollar sign; macro names longer than one character must be parenthesized. For example, the following are valid macro invocations:

```
$(CFLAGS)
$2
$(X)
$X
```

The last two invocations are identical.

When *make* encounters a macro invocation in a dependency line or command line of a makefile, it replaces it with the character string associated with the macro. For example, suppose that the macro **OBJECTS** is associated with the string *a.o b.o c.o d.o*. Then the dependency entries:

```
prog.exe: prog.o a.o b.o c.o d.o
          ln prog.o a.o b.o c.o d.o
a.o b.o c.o d.o: defs.h
```

within a makefile could be abbreviated as:

```
prog.exe: prog.o $(OBJECTS)
          ln prog.o $(OBJECTS)
$(OBJECTS): defs.h
```

There are three special macros: **\$\$**, **\$***, and **\$@**. **\$\$** represents the dollar sign. The other two are discussed below.

2.2.2 Defining macros in a makefile

A macro is defined in a makefile by a line consisting of the macro name, followed by the character '=', followed by the character string to be associated with the macro.

For example, the macro `OBJECTS`, used above, could be defined in the makefile by the line

```
OBJECTS = a.o b.o c.o d.o
```

A makefile can contain any number of macro definition entries. A macro definition must appear in the makefile before the lines in which it is used.

2.2.3 Defining macros in a command line

A macro can be defined in the command line that starts *make*. The syntax for a command line definition has the following form:

```
mac=str
```

where *mac* is the name of the macro, and *str* is its value.

str cannot contain spaces or tabs.

For example, the following command assigns the value *-DFLOAT* to the macro *CFLAGS*:

```
make CFLAGS=-DFLOAT
```

The assignment of a value to a macro in a command line overrides an assignment in a makefile statement.

2.2.4 Macros used by built-in rules

make has two macros, *CFLAGS* and *AFLAGS*, that are used by the built-in rules. These macros by default are assigned the null string. This can be overridden by a macro definition entry in the makefile.

For example, the following would cause *CFLAGS* to be assigned the string *"-T"*:

```
CFLAGS = -T
```

These macros are discussed below in the description of built-in rules.

2.2.5 Special macros

Before issuing any command, two special macros are set: *\$@* is assigned the full name of the target file to be made, and *\$** is the name of the target file, without its extension. Unlike other macros, these can only be used in command lines, not in dependency lines.

For example, suppose that the files *x.c*, *y.c*, and *z.c* need to be compiled using the option *"-DFLOAT"*. The following dependency entry could be used:

```
x.o y.o z.o:  
cc -DFLOAT $*.c
```

When *make* decides that *x.o* needs to be recreated from *x.c*, it will assign *\$** the string *"x"*, and the command


```
cc -DFLOAT x.c
```

will be executed. Similarly, when *y.o* or *z.o* is made, the command *cc -DFLOAT y.c* or *cc -DFLOAT z.c* will be executed.

The special macros can also be used in command lines associated with rules. In fact, the *\$@* macro is primarily used by rules. We'll discuss this more in the description of rules, below.

2.3 Rules

In section 1, we presented the basic features of rules: what they are and how they are used. We also noted that rules could be defined in the makefile and that some rules are built into *make*. In the following paragraphs, we describe how rules are defined in a makefile and list the built-in rules.

2.3.1 Rule definition

A rule consists of a source extension, target extension, and command list. In a makefile, an entry defining a rule consists of a line defining the two extensions, followed by lines containing the commands.

The line defining the extensions consists of the source extension, immediately followed by the target extension, followed by a colon.

All command lines associated with a rule must begin with a tab character. The first line following the extension line that doesn't begin with a tab terminates the commands for the rule.

For example, the following rule defines how to create a file having extension *.rel* from one having extension *.c*:

```
.c.rel:
    cc -o $@ $*.c
```

The first line declares that the rule's source and target extension are *.c* and *.rel*, respectively.

The second line, which must begin with a tab, is the command to be executed when a *.rel* file is to be created using the rule.

Note the existence of the special macros *\$@* and *\$** in the command line. Before the command is executed to create a *.rel* target file using the rule, the macro *\$@* is replaced by the full name of the target file, and the macro *\$** by the name of the target, less its extension.

Thus, if *make* decides that the file *x.rel* needs to be created using this rule, it will issue the command

```
cc -o x.rel x.c
```

If a rule defined in a makefile has the same source and target extensions as a built-in rule, the commands associated with the

makefile version of the rule replace those of the built-in version. For example, the built-in rule for creating a *.o* file from a *.c* file looks like this:

```
.c.o:
    cc $(CFLAGS) $*.c
```

If you want the rule to generate an assembly language listing, include the following rule in your makefile:

```
.c.o:
    cc $(CFLAGS) -a $*.c
    as -ZAP -l $*.asm
```

2.3.2 Built-in rules

The following rules are built into *make*. The order of the rules is important, since *make* searches the list beginning with the first one, and applies the first applicable rule that it finds.

```
.c.o:
    cc $(CFLAGS) -o $@ $*.c

.c.obj:
    cc $(CFLAGS) $*.c
    obj $*.o $@

.asm.obj:
    as $(AFLAGS) $*.asm
    obj $*.o $@

.asm.o:
    as $(AFLAGS) -o $@ $*.asm
```

The two macros *CFLAGS* and *AFLAGS* that are used in the built-in rules are built into *make*, having the null character string as their values. To have *make* use other options when applying one of the built-in rules, you can define the macro in the makefile.

For example, if you want the options *-T* and *-DDEBUG* to be used when *make* applies the *.c.o* rule, you can include the line

```
CFLAGS = -T -DDEBUG
```

in the makefile. Another way to accomplish the same result is to redefine the *.c.o* rule in the makefile; this, however, would use more lines in the makefile than the macro redefinition.

2.4 Commands

In this section we want to discuss the execution of operating system commands by *make*.

2.4.1 Allowed commands

A command line in a dependency entry or rule within a makefile can specify any command that you can enter at the keyboard. This includes batch commands, commands built into the operating system, and commands that cause a program to be loaded and executed from a disk file.

2.4.2 Logging commands and aborting make

Normally, before *make* executes a command, it writes the command to its standard output device; and when the command terminates, *make* halts if the command's return code was non-zero. Either or both of these actions can be suppressed for a command, by preceding the command in the makefile with a special character:

- @ Tells *make* not to log the command;
- Tells *make* to ignore the command's return code.

For example, consider the following dependency entry in a makefile:

```
prog.exe: a.o b.o c.o d.o
  ln -o prog.exe a.o b.o c.o d.o -lc
  @echo all done
```

When the *echo* command is executed, the command itself won't be logged to the console.

2.4.3 Long command lines

Makefile commands that start a Manx program, such as *cc*, *as*, or *ln*, or that start a program created with *cc*, *as*, *ln*, and *c.lib*, can specify a command line containing up to 2048 characters.

For example, if a program depends on fifty modules, you could associate them with the macro *OBJECTS* in the makefile, and also include the dependency entry

```
prog.exe: $(OBJECTS)
  ln -o prog.exe $(OBJECTS) -lc
```

This will result in a very long command line being passed to *ln*.

In the next section we will describe how *OBJECTS* could be defined.

For the execution of other commands, the command line can contain at most 127 characters.

2.5 Makefile syntax

We've already presented most of the syntax of a makefile; that is, how to define rules, macros, and dependencies. In this section we want to present two features of the makefile syntax not presented elsewhere:

comments and line continuation.

2.5.1 Comments

make assumes that any line in a makefile whose first character is '#' is a comment, and ignores it. For example:

```
#
# the following rule generates an 8080 object module
# from a C source file:
#
.c.o80:
    cc80 -o cc.tmp $*.c
    as80 -ZAP -o $*.o80 cc.tmp
```

2.5.2 Line continuation

Many of the items in a makefile must be on a single line: a macro definition, the file dependency information in a dependency entry, and a command that *make* is to execute must each be on a single line.

You can tell *make* that several makefile lines should be considered to be a single line by terminating each of the lines, except the last, with the backslash character, '\'. When *make* sees this, it replaces the current line's backslash and newline, and the next line's leading blanks and tabs by a single blank, thus effectively joining the lines together.

The maximum length of a makefile line after joining continued lines is 2048 characters.

For example, the following macro definition equates OBJ to a string consisting of all the specified object module names.

```
OBJ = printf.o fprintf.o format.o\
    scanf.o fscanf.o scan.o\
    getchar.o getc.o
```

As another example, the following dependency entry defines the dependence of *driver.lib* on several object modules, and specifies the command for making *driver.lib*:

```
driver.lib: driver.o printer.o \
    in.o \
    out.o
    lb driver.lib driver.o\
    printer.o \
    in.o out.o
```

This second example could have been more cleanly expressed using a macro:

```

DRIVOBJ= driver.o printer.o\
        in.o out.o
driver.lib $(DRIVOBJ)
lb driver.lib $(DRIVOBJ)

```

This was done to show that dependency lines and command lines can be continued, too.

2.6 Starting make

You've already seen how *make* is told to make a single file. Entering

```
make filename
```

makes the file named *filename*, which must be described by a dependency entry in the makefile. And entering

```
make
```

makes the first file listed as a target file in the first dependency entry in the makefile.

In both of these cases, *make* assumes the makefile is named 'makefile' and that it's in the current directory on the default drive.

In this section we want to describe the other features available when starting *make*.

2.6.1 The command line

The complete syntax of the command line that starts *make* is:

```
make [-n] [-f makefile] [-a] [macro=str] [file1] [file2] ...
```

Square brackets indicate that the enclosed parameter is optional.

The parameters *file1*, *file2* ... are the names of the files to be made. Each file must be described in a dependency entry in the makefile. They are made in the order listed on the command line.

The other command line parameters are options, and can be entered in upper or lower case. Their meanings are:

- n Suppresses command execution. *make* logs the commands it would execute to its standard output device, but doesn't execute them.
- f makefile Specifies the name of the makefile
- a Forces *make* to make all files upon which the specified target files directly or indirectly depend, and to make the target files, even those that it considers current.

MACRO=str

Creates a macro named MACRO, and assigns *str* as its value.

2.6.2 make's standard output

make logs commands and error messages to its standard output device. This can be redirected in the standard way. For example, to make the first target file in the first dependency entry and log messages to the file *out*, enter

```
make >out
```

The standard input and output devices of programs started by *make* are set as they are for *make* itself, unless one or both of them are explicitly redirected in the command that starts the program.

2.7 Executing commands

When *make* decides that a command needs to be executed, it executes it immediately, and waits for the command to finish. It activates a command whose code is contained in a disk file by issuing an *exec* function call. It activates DOS built-in commands and batch commands by calling the *system* function, which causes a new copy of the command processor to be loaded. Thus, to use *make*, your system must have enough memory for DOS, *make*, and whatever programs are loaded by *make* to be in memory simultaneously.

2.8 Differences between the Manx and UNIX 'make' programs

The Manx *make* supports a subset of the features of the UNIX *make*. The following comments present features of the UNIX *make* that aren't supported by the Manx *make*.

- * The UNIX *make* will let you make a file that isn't defined as a target in a makefile dependency entry, so long as a rule can be applied to create it. The Manx *make* doesn't allow this. For example, if you want to create the file *hello.o* from the file *hello.c* you could say, on UNIX

```
make hello.o
```

even if *hello.o* wasn't defined to be a target in a makefile dependency entry. With the Manx *make*, you would have to have a dependency entry in a makefile that defines *hello.o* as a target.

- * The UNIX *make* supports the following options, which aren't supported by the Manx *make*:

```
p, i, k, s, r, b, e, m, t, d, q
```

The Manx *make* supports the option '-a', which isn't supported by the UNIX *make*.

- * The special names *.DEFAULT*, *.PRECIOUS*, *.SILENT*, and *.IGNORE* are supported only by the UNIX *make*.
- * Only the UNIX *make* allows the makefile to be read from *make*'s standard input.

- * Only the UNIX *make* supports the special macros \$<, \$?, and \$%, and allows an upper case D or F to be appended to the special macros, which thus modifies the meaning of the macro.
- * Only the UNIX *make* requires that the suffixes for additional rules be defined in a .SUFFIXES statement.
- * Only the UNIX *make* allows macros to be defined on the command line that activates *make*.
- * Only the UNIX *make* allows a target to depend on a member of a library or archive.

3. Examples

3.1 First example

This example shows a makefile for making several programs. Note the entry for *arc*. This doesn't result in the generation of a file called *arc*; it's just used so that we can generate *arcv* and *mkarcv* by entering *make arc*.

```

#
# rules:
#
.c.o80:
    cc80 -DTINY -o $@ $*.c
#
# macros:
#
OBJ=make.o parse.o scandir.o dumptree.o rules.o command.o
#
# dependency entry for making make:
#
make.com: $(OBJ) cntlc.o envcopy.o
    ln -o make.com $(OBJ) envcopy.o cntlc.o -lc
#
# dependency entries for making arcv & mkarcv:
#
arc: mkarcv.com arcv.com
    @echo done
mkarcv.com: mkarcv.o
    ln -o mkarcv.com mkarcv.o -lc
arcv.com : arcv.o
    ln -o arcv.com arcv.o -lc
#
# dependency entries for making CP/M-80 versions of arcv & mkarcv:
#
mkarcv80.com: mkarcv.o80
    ln80 -o mkarcv80.com mkarcv.o80 -lt -lc
arcv80.com: arcv.o80
    ln80 -o arcv80.com arcv.o80 -lt -lc

```

\$(OBJ): libc.h make.h

3.2 Second example

This example uses *make* to make a library, *my.lib*. Three directories are involved: the directory *libc* and two of its subdirectories, *sys* and *misc*. The C and assembly language source files are in the two subdirectories. There are makefiles in each of the three directories, and this example makes use of all of them. With the current directory being *libc*, you enter

```
make my.lib
```

This starts *make*, which reads the makefile in the *libc* directory. *make* will change the current directory to *sys* and then start another *make* program.

This second *make* compiles and assembles all the source files in the *sys* directory, using the makefile that's in the *sys* directory.

When the '*sys*' *make* finishes, the '*libc*' *make* regains control, and then starts yet another *make*, which compiles and assembles all the source files in the *misc* subdirectory, using the makefile that's in the *misc* directory.

When the '*misc*' *make* is done, the '*libc*' *make* regains control and builds *my.lib*. You can then remove the object files in the subdirectories by entering

```
make clean
```

3.2.1 The makefile in the '*libc*' directory

```
my.lib: sys.mk misc.mk
    del my.lib
    lb my.lib -f my.bld
    @echo my.lib done
```

```
sys.mk:
    cd sys
    make
    cd ..
```

```
misc.mk:
    cd misc
    make
    cd ..
```

```
clean:
    cd sys
    make clean
    cd ..
    cd misc
    make clean
    cd ..
```

3.2.2 Makefile for the 'sys' directory

```
REL=asctime.o bdos.o begin.o chmod.o croot.o csread.o ctime.o \
dostime.o dup.o exec.o excl.o execlp.o execv.o execvp.o \
fexec.o fexecl.o fexecv.o ftime.o getcwd.o getenv.o \
isatty.o localtime.o mkdir.o open.o stat.o system.o time.o \
utime.o wait.o dioctl.o ttyio.o access.o syserr.o
```

COPT=

HEADER=../header

.c.o:

```
cc $(COPT) -I$(HEADER) $*.c -o $@
sqz $@
```

.asm.o:

```
as $*.asm -o $@
sqz $@
```

all: \$(REL)

```
@echo sys done
```

clean:

```
del *.o
```

3.2.3 Makefile for the 'misc' directory

```
REL=atoi.o atol.o calloc.o ctype.o format.o malloc.o qsort.o \
sprintf.o sscanf.o fformat.o fscan.o
```

COPT=

HEADER=../header

.c.o:

```
cc $(COPT) -I$(HEADER) $*.c -o $@
sqz $@
```

.asm.o:

```
as $*.asm -o $@
sqz $@
```

all: \$(REL)

```
@echo misc done
```

ffformat.o: format.c

```
cc -I$(HEADER) -DFLOAT format.c -o fformat.o
```

fscan.o: scan.c

```
cc -I$(HEADER) -DFLOAT scan.c -o fscan.o
```

clean:

```
del *.o
```

NAME

obd65 - list object code

SYNOPSIS

obd65 <objfile>

DESCRIPTION

obd65 lists the loader items in an object file. It has a single parameter, which is the name of the object file.

NAME

`optint65` - pseudo-code optimizer

SYNOPSIS

`optint65 [-ZAP] [-o outfile] [-a] [-v] infile`

DESCRIPTION

optint65 optimizes the assembly language source that's generated by *cci*. The resulting code can then be assembled by *asi*.

infile is the name of the file whose assembly language source is to be optimized.

The *-ZAP* option tells *optint65* to delete the input file when the optimization is completed.

The *-o outfile* tells *optint65* to write the optimized code to the file named *outfile*. If this option isn't used, the optimized code is written to a file whose name is derived from that of the input file, by changing its extension to *.opt*.

The *-a* option tells *optint65* not to start *asi*. If this option isn't used, *optint65*, when done, starts *asi*, which assembles the optimized code and writes the resultant object code to a file. The name of this file is derived from the optimized code file by changing the extension to *.i*. In this default case, *asi*, when done, deletes the optimized code file.

The *-v* option tells *optint65* to display information about the optimizations that it performs.

NAME

ord65 - sort object module list

SYNOPSIS

ord65 [-v] [infile [outfile]]

DESCRIPTION

ord65 sorts a list of object file names. A library of the object modules that is generated from the sorted list by the Manx object module librarian will have a minimum number of 'backward references'; that is, global symbols that are defined in one module and referenced in a later module.

Since the specification of a library to the linker causes it to search the library just once, a library having no backward references need be specified just once when linking a program, and a library having backward references may need to be specified multiple times.

infile is the name of a file containing an unordered list of file names. These files contain the object modules that are to be put into a library. If *infile* isn't specified, this list is read from *ord65*'s standard input. The file names can be separated by space, tab, or newline characters.

outfile is the name of the file to which the sorted list is written. If it's not specified, the list is written to *ord65*'s standard output. *outfile* can only be specified if *infile* is also specified.

The *-v* option causes *ord65* to be verbose, sending messages to its standard error device as it proceeds.

NAME

sqz65 - squeeze an object library

SYNOPSIS

sqz65 file [outfile]

DESCRIPTION

sqz65 compresses an object module that was created by the Manx assembler.

The first parameter is the name of the file containing the module to be compressed. The second parameter, which is optional, is the name of the file to which the compressed module will be written.

If the output file is specified, the original file isn't modified or erased.

If the output file isn't specified, *sqz65* creates the compressed module in a file having a temporary name, erases the original file, and renames the output file to the name of the original file. The temporary name is derived from the input file name by changing its extent to *.sqz*.

If the output file isn't specified and an error occurs during the creation of the compressed module the original file isn't erased or modified.

LIBRARY GENERATION

Chapter Contents

Library generation libgen

1. Rewriting the functions 3

1.1 The start-up function 3

1.2 The `__main` function 4

1.3 The `Unbuffered` i/o functions 4

1.4 The standard i/o functions 'agetc' and 'aputc' 9

1.5 The `sbrk` heap management function 9

1.6 The `exit` and `__exit` functions 9

2. Building the libraries 10

3. Function descriptions 11

Library Generation

The Aztec CG65 functions are provided in source form. Before you can create programs that use them, you will have to create object module libraries of them, after making any necessary modifications.

In the following discussion, we assume that you have installed Aztec CG65 in a set of subdirectories, as directed in the Tutorial chapter. We also assume that your system has a *make* program maintenance program that is UNIX compatible; this program, under direction of "makefiles" provided with Aztec CG65, will control the compilation and assembly of library modules and the generation of the libraries. For systems whose standard software doesn't include *make*, we will provide the Aztec *make* with your Aztec CG65 package, if one is available; otherwise, the release document will describe the procedure for creating the libraries.

The description of the Aztec *make* is in the Utility Programs chapter.

1. Rewriting the functions

Many of the functions provided with this package will run, without modification, on any 65xx-based system. Some, however, may need to be rewritten for use on different systems. We've included the source for the Apple // versions of these functions, which you can modify for use on your system.

The functions that may need to be rewritten are:

- * The start-up function;
- * The `__main` function;
- * The unbuffered i/o functions;
- * The standard i/o functions *agetc* and *aputc*.

1.1 The start-up function

The start-up function is the first routine to be executed when the program is started. It sets up pointers, moves the copy of the initialized data segment from ROM to RAM, clears the uninitialized data segment, and jumps to the program's *main* function. The startup function is named *.begin*; its source is in the file *rom.a65*, in the *rom.arc* archive.

The following paragraphs describe some changes that you might want to make to *rom.a65*:

- * *rom.a65* contains a statement that creates a 2kb area for the program's pseudo stack in the uninitialized data area. You can change this statement to, for example, change the size of this area, or to place pseudo stack outside of the initialized data area, or ...
- * *rom.a65* contains statements that define the boundaries of a program's 'heap'; that is, the area of memory from which buffers are dynamically allocated. By default, this area is 1 kb long, and immediately follows the space reserved for the program's uninitialized data and, if present, its overlays. You can change these statements to, for example, change the size of the heap, or to place it in some other section of memory, or ...
- * The 65xx has three fields at the top of memory that contain pointers to routines that handle power-up/reset, nmi interrupt, and irq interrupt. *hex65* can optionally generate hex records that initialize the 65xx power-up/reset and interrupt vectors; when it does so, it sets the address of the global symbol *.begin* in the power-up/reset vector, *.nmi* in the nmi vector, and *.irq* in the irq vector. You already know that *.begin* is in *rom.a65*. It also contains the directives that define *.irq* and *.nmi*; no code, just the definition directives. So if your program is going to handle these interrupts, you must either add the code to *rom.a65* or remove these directives from *rom.a65* and put them and the interrupt-handling code in another module.

1.2 The *__main* function

The *__main* function, whose source is in *umain.c* within the *rom.arc* archive, acts as an interface between the *.begin* and *main* functions. In the supplied version, *__main* just calls *main*, passing null values for *main*'s *argc* and *argv* parameters. You may want to modify this function, to initialize the program's stdin, stdout, and stderr devices, to handle i/o redirection, to pass command line arguments to *main* via the *argc* and *argv* parameters, ...

1.3 The Unbuffered i/o functions

There are two classes of UNIX-compatible i/o functions: standard and unbuffered. The unbuffered i/o functions are system dependent, and the standard i/o functions call the unbuffered. Aztec CG65 contains the Apple ProDOS versions of these functions; so you must rewrite those that your functions call, and those that are called by the standard i/o functions that your functions call.

The unbuffered i/o functions are:

open	creat	close	read	write
lseek	rename	unlink	ioctl	isatty

Descriptions of the unbuffered i/o functions are in the "System Independent Functions" and "Library Functions Overview" chapters. The following paragraphs present additional information that may be of use when writing your own versions of these functions.

1.3.1 File descriptors

Associated with each file or device that is open for unbuffered i/o is a positive integer called a "file descriptor". A file descriptor is one of the parameters that is passed to an unbuffered i/o function; it defines the file or device on which the i/o is to be performed. There's usually a limited number of file descriptors, which of course limits the number of files and/or devices that can be simultaneously open for i/o.

1.3.1.1 When there's lots of files and devices...

If a system supports disk files and/or supports more devices than file descriptors, the file descriptors must be dynamically allocated. That is, before i/o with a file or device can begin, a function must be called that assigns a file descriptor to it; and when the i/o is done another function must be called to de-assign the file descriptor. In this case, a table is usually provided that has entries defining the status of each file descriptor and that is accessible to all the unbuffered i/o functions. Here's how the unbuffered i/o functions make use of the table:

- * *open* and *creat* prepare a file or device for unbuffered i/o. They scan the table for an unused entry, and initialize the entry with information about the file or device. For example, the entry for an open device might contain the device's address; that for an open file might contain the file's current position and access mode. As the file descriptor for the opened file or device, *open* and *creat* return the entry's index into the table.
- * *read*, *write*, *lseek*, *ioctl*, and *isatty* perform operations on, and determine the status of, an open file or device. The file descriptor of the file or device is one of the parameters passed to them. They examine the file descriptor's table entry for information about the file or device.
- * *close* completes i/o to the open file or device having a specified file descriptor. Most of the operations that *close* performs depend on the particular file or device; but it always marks the descriptor's table entry as being unused.
- * *unlink* and *rename* don't use the file descriptor table at all.

1.3.1.2 When only devices are supported..

If programs access just devices (i.e. not files), if there are fewer devices than file descriptors, and if your programs make limited use of the standard i/o functions (as defined below), you can simplify the unbuffered i/o functions by doing away with the file descriptor table, hard-coding the assignment of devices and file descriptors into the unbuffered i/o functions, and leaving *open*, *creat*, and *close* as mere stubs that simply return when called.

For example, you could code into the *write* function the fact that file descriptor 5 is associated with a printer at a certain address. Then to write to the printer, a program could simply issue a call to *write*, telling it to write to file descriptor 5. It wouldn't have to first call *open* or subsequently call *close*.

1.3.1.3 Pre-assigned file descriptors

By convention, file descriptors 0, 1, and 2 are pre-assigned to the system console, even when all other file descriptors are dynamically assigned. To perform an unbuffered i/o operation on the console, a program simply calls the appropriate function, specifying one of these file descriptors; it need not first call *open* or subsequently call *close*.

Some systems allow the operator to redirect file descriptors 0 and 1 to other files and/or devices, by specifying special operands on the command line that starts a program. This is done by inserting a special function between the startup routine and the user's *main* function. If any redirection operands are found in the command line, this special function closes the specified file descriptor by calling *close* and reopens it to the new file or device by calling *open*. By convention, the command line operand to redirect file descriptor 0 consists of "<" followed by the file or device name. The command line operand to redirect file descriptor 1 consists of ">" or ">>" followed by the file or device name. ">" causes a new file to be created. ">>" causes a file to be appended to, if it already exists, or to be created, if it doesn't exist.

1.3.2 Interaction of the standard i/o and unbuffered i/o functions

The standard i/o functions call the unbuffered i/o functions. Because of this, the standard i/o operations that a program will perform places implementation requirements on the unbuffered i/o functions. This section discusses those requirements, after first presenting general information on standard i/o file pointers and their relationship to unbuffered i/o file descriptors.

Before standard i/o can be performed on a file or device, an unbuffered i/o file descriptor must be assigned to it, and a standard i/o "file pointer" must be assigned to the file descriptor. The assignment of a file pointer and file descriptor can be done dynamically, by calling the standard i/o *fopen* function. Three file pointers, named *stdin*, *stdout*, and *stderr*, are pre-assigned to file

descriptors 0, 1, and 2; these file descriptors in turn are pre-assigned to the console.

When a program calls a standard i/o function, it often must pass a file pointer, which identifies the file or device on which i/o is to be performed. There are a special set of standard i/o functions for accessing stdin, stdout, and stderr: for these, the file pointer isn't passed, since the functions know what file pointer is being accessed.

1.3.2.1 Supporting the standard i/o *fopen* and *fclose* functions

The dynamic assignment of a file pointer and file descriptor to a file or device is done by the *fopen* function. This function selects a file pointer for the file or device and then calls the unbuffered i/o *open* function, which selects a file descriptor.

If programs call *fopen*, you must implement the unbuffered i/o *open* function, and *open* must return the file descriptor that's associated with the file or device. This requirement (for a functional *open* when *fopen* is called) must be met even if file descriptors are pre-assigned to devices; *open* in this case could be very simple, just searching a table for a device name and returning the associated file descriptor.

Conversely, the use of the standard i/o functions to access those devices that don't first have to be *opened* (i.e. stdin, stdout, and stderr) places no requirements on *open*. In particular, if file descriptors are pre-assigned to devices and *open* simply returns when called, programs can still call the standard i/o functions to access the devices associated with the stdin, stdout, and stderr file pointers.

The standard i/o function *fclose* calls the unbuffered i/o function *close*. Thus, if programs call *fclose*, you must implement a *close* function. If assignments of devices to file descriptors is hard-coded, *close* can usually just return the value 0, since nothing special (such as calling the operating system to close an open file or deallocating a file descriptor) needs to be done.

1.3.2.2 Supporting the standard i/o input and output functions

If programs call any of the standard i/o input functions, you must implement the unbuffered i/o *read* function. And if they call any of the standard i/o output functions, you must implement the *write* function.

1.3.2.3 Supporting the standard i/o *fseek* function

If programs will call the standard i/o *fseek* function, you must implement the unbuffered i/o *lseek* function, since *fseek* calls *lseek*.

1.3.2.4 Standard i/o and the *isatty* function

If programs call any standard i/o functions, you must implement the unbuffered i/o function *isatty*. The standard i/o functions call this function to decide whether their i/o to a file or device should be

buffered or unbuffered.

This use of the word "unbuffered" in describing standard i/o might be a little confusing, since the use of the expression "unbuffered i/o functions" to describe one set of i/o functions implies that the other set, the "standard i/o functions", are buffered. Nevertheless, a standard i/o stream can be either buffered or unbuffered: if buffered, data that's exchanged between user-written functions and the unbuffered i/o functions passes through a buffer; if unbuffered, data doesn't pass through a buffer.

For a given file descriptor, *isatty* should return non-zero if standard i/o to the device associated with the file descriptor is to be buffered, and zero if it is to be unbuffered.

For example, *isatty* should probably return non-zero for a file descriptor that's associated with the system console and zero for file descriptors associated with files; it could return either zero or non-zero for other devices, such as printers, depending on your system's requirements.

1.3.3 Error codes

We've presented most of the factors you should consider when writing your unbuffered i/o functions. In this section we want to list error codes that the functions could return in the global *errno*.

open error codes:

ENOENT File does not exist and O_CREAT wasn't specified.
EEXIST File exists, and O_CREAT+O_EXCL was specified.
EMFILE Invalid file descriptor passed to *open*.

close error codes:

EBADF Bad file descriptor passed to *close*.

creat error codes:

EMFILE All file descriptors are in use.

lseek error codes:

EBADF Invalid file descriptor
EINVAL Offset parameter is invalid, or the requested position is before the beginning of the file.

read error codes:

EBADF Invalid file descriptor

write error codes:

EBADF Invalid file descriptor
EINVAL Invalid operation; i.e. writing not allowed.

1.4 The standard i/o functions 'agetc' and 'aputc'

The characters used to terminate lines of text differ from system to system. On UNIX, it's the newline (linefeed) character, '\n'. On the Apple //, it's carriage return, '\r'. On CPM, it's carriage return-line feed. In order to allow programs to access files of text in a system-independent manner, the standard i/o functions *agetc* and *aputc* are provided: *agetc* reads a character from the standard input channel, translating the line termination sequence into '\n'. *aputc* writes a character to the standard output channel, translating '\n' to the line termination sequence.

The following standard i/o functions call *agetc* and *aputc*:

scanf	fscanf	printf	sprintf
getchar	gets	fgets	
putchar	puts	fputs	

Hence, if you intend to write programs that access text and the line termination sequence on your system differs from that on the Apple // (that is, it isn't carriage return), you'll have to modify *agetc* and *aputc*.

The source for these functions are in the files *agetc.c* and *aputc.c*, within the *stdio.arc* archive. If you followed our recommendations for installing Aztec CG65, dearchived versions are also in the *STDIO* subdirectory of the *LIB* directory.

1.5 The *sbrk* heap management function

sbrk provides an elementary means of allocating and deallocating space from a program's heap. *sbrk* is called by the more sophisticated heap-allocation functions (*malloc*, etc), and *malloc* is called by the standard i/o functions; thus, if your programs call *malloc* or the other high-level heap management functions, or if they call the standard i/o functions, you will need an *sbrk* function.

You probably won't have to modify *sbrk*, since the most system-dependent code (which defines the boundaries of the heap) is in the *startp* routine.

A description of *sbrk*'s calling sequence is appended to this chapter.

1.6 The *exit* and *__exit* functions

exit and *__exit* are called to terminate the execution of a program. They aren't usually called by ROM-based programs, since such programs usually don't terminate.

They are called, however, by RAM-based programs that are running in an operating system environment, since these programs usually do terminate.

When these functions are needed, you will have to modify `__exit`, since it must return to the operating system. But you can probably use `exit` as is, since it closes open files and devices in a system-independent way and then calls `__exit`.

Descriptions of the calling sequences to `exit` and `__exit` are appended to this chapter.

2. Building the libraries

Once you've made modifications to the supplied unbuffered i/o functions, you can build your libraries. We recommend that you create the following libraries:

<i>c.lib</i>	General purpose functions (<i>cg65</i> -compiled)
<i>ci.lib</i>	General purpose functions (<i>cci</i> -compiled)
<i>m.lib</i>	Floating point functions (<i>cg65</i> -compiled)
<i>mi.lib</i>	Floating point functions (<i>cci</i> -compiled)

To simplify the creation of these libraries, Aztec CG65 contains several "makefiles" that give directions to the *make* program maintenance utility, and a few files that give directions to the *lb* object module librarian. If you followed our recommendations for installing Aztec CG65, each of the LIB directory's subdirectories contains a makefile that causes *make* to compile and assemble the subdirectory's source files. There is a makefile in the LIB directory that can be used on systems having lots of memory, to have *make* first generate each subdirectory's object modules and then make a library.

Before you can generate the libraries, you must do several things:

1. In each makefile, modify the rules that define how to convert a C source file to an object module, so that the command that starts the compiler uses a `+G` option that correctly defines zero-page usage on your system.
2. Modify the *zpage.h* file in the INCLUDE directory. This file defines the use of zero page for assembly language modules.
3. You've probably created a subdirectory of the LIB directory, a subdirectory that contains your own unbuffered i/o modules. In this subdirectory you should create a makefile that tells *make* how to generate object modules from your files.
4. In the LIB directory are four files (*c.bld*, *ci.bld*, *m.bld*, and *mi.bld*), each of which tells *lb* how to create a library. *c.bld* and *ci.bld* are used for generating ProDOS versions of *c.lib* and *ci.lib*, so you will need to modify these files. Some of the changes that you'll need to make are these: (1) instead of including the Apple // startup routine *crt0.r* that's in the PRODOS directory, include the 65xx ROM startup routine *rom1.r* that's in the ROM directory; (2) instead of including the

ProDOS *__main* routine that's in the *shmain.r* module in the PRODOS directory, include the 65xx ROM *__main* routine that's in the *umain.r* module in the ROM directory; (3) replace the ProDOS unbuffered i/o modules with your own.

5. The environment variable INCL65 must be set to the name of the INCLUDE directory; that is, to the name of the directory that contains the include files. The command to do this varies from system to system; on PCDOS, it's the *set* command.
6. If you have a RAM disk, you can speed up the library-generation process by defining it using the CCTEMP environment variable. For more information, see the description of CCTEMP in the Compiler chapter.

You are now ready to create the libraries. If your system has lots of memory, you can create a library setting the default or current directory to the LIB directory starting *make*, passing to it the name of the library you want created. For example, to create *c.lib*, you would enter:

```
make c.lib
```

For non-UNIX systems, a special makefile (named *makepc*) is provided in *libmake.arc* that should be used in place of the standard makefile (named *makefile*). To make *c.lib* using *makepc*, type

```
make -f makepc c.lib
```

Once started, *make* will activate several other copies of *make*, each of which will compile and assemble the files in one of LIB's subdirectories; it will then start *lb*, which will make the specified library from the object modules that are in the subdirectories, as directed by the appropriate *.bld* file.

If your system doesn't have lots of memory (if there's not enough memory, *make* will abort with the message "EXEC failure"), you can create and execute batch files that will generate the libraries. A batch file will first, for each subdirectory, make that subdirectory the default or current directory and then activate *make*, using the command *make rel* to make *cg65*-compiled modules, or *make int* to make *cci*-compiled modules. The batch file will then activate *lb*, passing to it the name of the appropriate *.bld* file.

3. Function descriptions

The System Independent Functions chapter presents the calling sequences of most of the functions that are discussed in this chapter. The remainder of this chapter presents the calling sequences of the other functions.

NAME

`sbrk`

SYNOPSIS

`void *sbrk(size)`

DESCRIPTION

sbrk provides an elementary means of allocating and deallocating space from the heap. More sophisticated buffer management schemes can be built using this function; for example, the standard functions *malloc*, *free*, etc call *sbrk* to get heap space, which they then manage for the calling functions.

sbrk increments a pointer, called the 'heap pointer', by *size* bytes, and, if successful, returns the value that the pointer had on entry. Initially, the heap pointer points to the base of the heap. *size* is a signed *int*; if it is negative, the heap pointer is decremented by the specified amount and the value that it had on entry is returned. Thus, you must be careful when calling *sbrk*: if you try to pass it a value greater than 32K, *sbrk* will interpret it as a negative number, and decrement the heap pointer instead of incrementing it.

SEE ALSO

The functions *malloc*, *free*, etc, implement a dynamic buffer-allocation scheme using the *sbrk* function. See the Dynamic Buffer Allocation section of the Library Functions Overview chapter for more information.

The standard i/o functions usually call *malloc* and *free* to allocate and release buffers for use by i/o streams. This is discussed in the Standard I/O section of the Library Functions Overview.

Your program can safely mix calls to the *malloc* functions, the standard i/o functions, and *sbrk*, as long as the calls to *sbrk* don't decrement the heap pointer. Mixing *sbrk* calls that decrement the heap pointer with calls to the *malloc* functions and/or the standard i/o functions is dangerous and probably shouldn't be done by normal programs.

ERRORS

If an *sbrk* call is made that would result in the heap pointer passing beyond the end of the heap, *sbrk* returns -1, after setting the global integer *errno* to the symbolic value ENOMEM.

NAME

`exit`, `__exit`

SYNOPSIS

`exit(code)`

`__exit(code)`

DESCRIPTION

These functions cause a program to terminate and control to be returned to the operating system.

code is returned to the operating system, as the program's termination code.

exit and *__exit* differ in that *exit* closes all files opened for standard and unbuffered i/o, while *__exit* doesn't.

TECHNICAL INFORMATION

Chapter Contents

Technical Information tech

1. Memory Organization 4

2. Overlays 7

3. Interfacing to Assembly Language 14

4. Object Code Format 18

5. The Pseudo Stack 29

Technical Information

This chapter discusses technical topics, and topics that couldn't be conveniently discussed elsewhere.

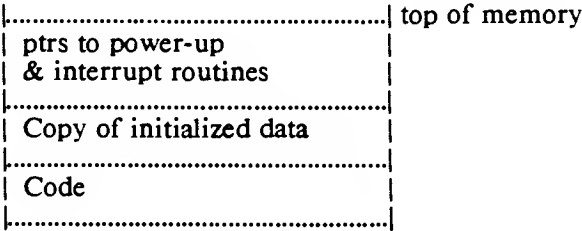
It's divided into the following sections:

1. *Memory Organization.* Discusses the factors that affect the memory organization of a program.
2. *Overlays.* Describes overlays: what they are, and how they're used.
3. *Mixing Assembler and C Routines.* Describes how to interface assembly language routines with C routines.
4. *Object Code Format.* Describes the format of object modules and libraries.
5. *The pseudo stack.* Describes the pseudo stack that is used by programs that have been created by Aztec CG65.

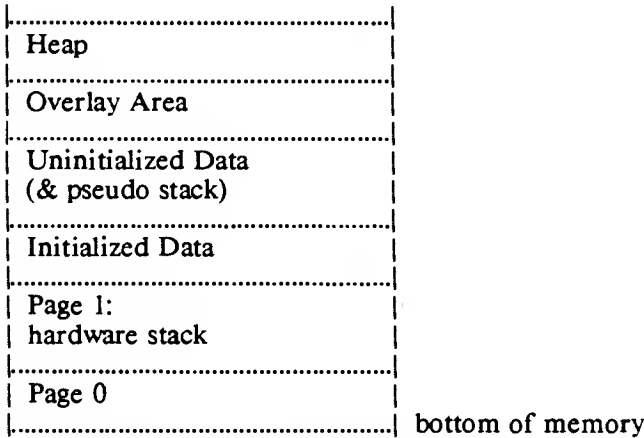
1. Memory Organization

A ROM program is organized into several sections. The linker lets you specify the position of some of these sections, but for a ROM system they are frequently positioned as follows:

ROM



RAM



The following paragraphs discuss these areas.

1.1 ROM sections

1.1.1 The code area

The code area contains the executable code for a program's root segment (i.e. for its non-overlay segment).

1.1.2 Copy of initialized data

A program's initialized data area resides in RAM and contains global and static variables that are assigned an initial value. For example, if the following statement occurs outside all functions, then the variable *var* would be placed in the program's initialized data area:


```
int var=1;
```

Since the initialized data segment resides in RAM, its contents will initially be unknown when the system is turned on. The Aztec CG65 startup routine sets up this segment, using the copy of the initialized data area that resides in ROM above the code segment.

The ROM-resident copy of the RAM-resident initialized data area is created automatically by *hex65* when it translates the memory image of the program, as generated by the linker, into Intel hex records.

1.1.3 Pointers to the power-up and interrupt routines

These pointers define the locations to which the 65xx will transfer control when power is turned on, when the processor is reset, or when an interrupt occurs. By default, they are generated by *hex65* when it converts the memory image of the program, as created by the linker, into Intel hex records. *hex65* sets the addresses of the *.nmi*, *.begin*, and *.irq* routines in the *nmi* power-up/reset, and *irq* fields, respectively.

1.2 RAM sections

1.2.1 The Initialized Data Area

This area was discussed above.

1.2.2 The Uninitialized data area

This area contains the global and static variables that aren't assigned an initial value.

It also contains the area in which the program's pseudo stack is placed. The "pseudo stack" is a stack simulated by the Aztec CG65 software to get around the limitations of the 65xx hardware stack (the hardware stack can be at most 256 bytes long).

When a program starts, the Aztec CG65 startup routine automatically clears the uninitialized data area.

1.2.3 The Overlay Area

A program's overlays are loaded into the overlay area. The size of this area is set when you link the program's root segment, to the sum of the values specified in the +C and +D options. By default, these options are set to zero, resulting in an overlay area that is zero bytes long.

For more information on overlays, see the Overlay section of this chapter.

1.2.4 The Heap

The heap is the area of memory from which buffers are dynamically allocated.

As defined by the Aztec CG65 startup routine, the heap is 1 kb long.

1.3 Symbols related to Program Organization

The following global symbols are related to program organization. The symbols are given in the form that an assembly language program would use to access them. A C module can access the symbols by removing the appended underscore from the symbol name.

<code>__Corg__</code>	Name of the beginning of the program's code.
<code>__Cend__</code>	Name of the first byte beyond the program's executable code.
<code>__Dorg__</code>	Name of the beginning of the program's initialized data.
<code>__Dend__</code>	Name of the first byte beyond the program's initialized data.
<code>__Uorg__</code>	Name of the beginning of the program's uninitialized data.
<code>__Uend__</code>	Name of the first byte beyond the program's uninitialized data.
<code>__mbot__</code>	Name of a field containing a pointer to the beginning of the program's heap.
<code>__Top__</code>	Name of a field containing a pointer to the next byte to be allocated from the heap.
<code>__End__</code>	Name of a field containing a pointer to the end of the program's heap.

1.4 For more information

For more information on the positioning of a program's segments, see the Tutorial chapter and the Linker chapter's discussion of segment-positioning options.

2. Overlay Support

In order to allow you to run programs which are larger than the limited memory size of a microcomputer, Manx provides overlay support. To use this feature, you must rewrite the unbuffered i/o functions whose source is provided with Aztec CG65. This feature allows you to divide a program into several segments. One of the segments, called the root segment, is always in memory. The other segments, called overlays, reside on disk and are only brought into memory when requested by the root segment.

If an overlay is in memory when the root requests that another be loaded, the newly specified overlay replaces the first in memory.

Overlays can also be "nested"; that is, an overlay at one level can call another overlay nested one level deeper. However, an overlay cannot call an overlay which is at the same level.

Figure 1 shows a program, run as a single module, that can be logically divided into three segments. Figure 2 shows the same program run as an overlay. In figure 2, module 1 and module 2 occupy the same memory locations. A possible flow of control would be for the base routine to call module 1, module 1 then returns to the root and the root calls module 2, module 2 returns to the root and the root calls module 1 again. Module 1 then returns to the root and the root exits to the operating system.

Notice that all overlay segments must return to their caller and that overlays at the same level cannot directly invoke each other.

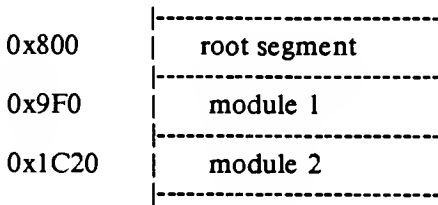


Figure 1

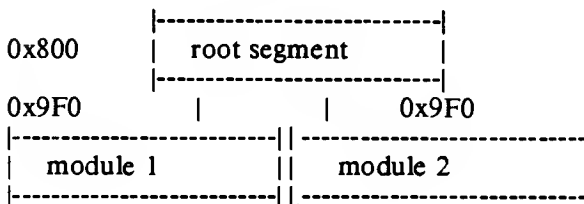


Figure 2

2.1 Calling an Overlay

A program segment (root or overlay) activates an overlay by calling the Manx-supplied function *ovloader*, which must reside in the root. The call has the form

```
ovloader(ovlyname, p1, p2, p3, ...)
```

where *ovlyname* is a pointer to a character string identifying the overlay name, and *p1*, *p2*, *p3*, ... are parameters that are to be passed to the overlay as its first, second, third, ... parameters.

ovloader derives the name of the file containing the overlay from the string pointed at by *ovlyname*, by appending the extension *.ovr* to it.

We provide you with the source to *ovloader*. When you compile it, you define the directories in which it will look for overlays: compiling it with the option *-DPATH* will cause it to search all directories specified in the *PATH* environment variable; compiling it without this option causes it to search just the current directory. If you create an overlaid program that will run under ProDOS outside of the SHELL environment or that will run under DOS 3.3, you must use a version of *ovloader* for it that looks for overlays in just the current directory, since environment variables are only available to programs running in the SHELL environment.

Each overlay must contain a function named *ovmain*, which you must write and which can be different for each overlay, and must also contain the Manx-supplied function named *ovbgn*. When an overlay is loaded, *ovloader* calls the overlay's *ovbgn* function, which in turn calls the overlay's *ovmain* function, passing to it the second, third, ... arguments that were passed to *ovloader*.

When *ovmain* completes its processing, it simply returns. *ovloader* then returns to the caller, returning as its value the value that was returned by *ovmain*.

An overlay can access any global functions and variables that are defined in the root segment and in the overlays that are currently active. For example, if the root calls overlay *ovly1*, which calls overlay *ovly11*, which calls overlay *ovly111*, then *ovly111* can access the global variables and functions that are defined in the root, in the overlays *ovly1* and *ovly11*, and in itself. But if the root also calls overlay *ovly2*, *ovly111* cannot access the global functions and variables that are in *ovly2*, since *ovly2* is not active when *ovly111* is.

2.2 Creating a root and its overlays

To create a root and its overlays, the linker must be run several times, once to create the root, and once for each overlay. Each program segment (root or overlay) will be placed in a separate disk file.

The root must be created first. When overlays are nested, an overlay that itself calls overlays must be linked before the overlays that it calls.

When creating a program segment (root or overlay) which calls an overlay, the option `-R` must be specified; this causes the linker to generate a symbol table for use in linking the called overlay, placing it in a file whose filename is the same as that of the first file specified in the command line and whose extent is `.rsm`. When an overlay is linked, the symbol table file of the program segment that calls the overlay must be included in the linkage of the overlay.

When the root module is linked, the linker has to reserve some space into which the overlay can be loaded. This is done using the `+C` and `+D` linker options, which define the amount of space needed for the overlay code and data, respectively. If overlays are nested, a called overlay is located in memory immediately following the calling overlay. The amount of space reserved for the overlays must be enough to hold the longest 'thread' of overlays.

2.3 Example 1: Non-nested Overlays

This example demonstrates overlay usage when the overlays are not nested. The root segment, which consists of the function *main* and any necessary run-time library routines, behaves as follows:

1. It calls the overlay *ovly1*, passing as a parameter a pointer to the string "first message".
2. It prints the integer value returned to it by *ovly1*;
3. It calls the overlay *ovly2*, passing a pointer to the string "second message";
4. It prints the integer value returned to it by *ovly2*.

The overlay *ovly1* consists of the function *ovly1*, the Manx function *ovbgn*, and any necessary run-time library routines. It prints the message "in *ovly1*" plus whatever character string was passed to it by *main*.

The overlay *ovly2* consists of the function *ovly2*, the function *ovbgn*, and any necessary run-time library routines. It prints the message "in *ovly2*", plus whatever character string was passed to it by *main*.

Here then is the *main* function:

```
main() {
    int a;

    a = ovloader("ovly1","first message");
    printf("in main. ovly1 returned %d\n", a);
    a = ovloader("ovly2","second message");
    printf("in main. ovly2 returned %d\n",a);
}
```

Here is *ovly1*:

```
ovmain(a)
char *a;
{
    printf("in ovly1. %s\n",a);
    return 1;
}
```

Here is *ovly2*:

```
ovmain(a)
char *a;
{
    printf("in ovly2. %s\n",a);
    return 2;
}
```

The following commands link the root (which is in the file *root.c*) and the overlays:

```
ln65 -R +C 4000 +D 1000 root.r ovloader.r -lc
ln65 ovly1.r ovbgn.r root.rsm -lc
ln65 ovly2.r ovbgn.r root.rsm -lc
```

The command to link the root reserves 0x4000 bytes for the overlay's code and 0x1000 bytes for it's data. Techniques for determining this value are discussed below.

When the segments are generated and the root activated, the following messages appear on the console:

```
in ovly1. first message.
in main. ovly1 returned 1.
in ovly2. second message.
in main. ovly2 returned 2.
```

2.4 Example 2: Nested Overlays

In this example, there are three segments: a root segment, *root*, and two overlays segments, *ovly1* and *ovly2*. *root* calls *ovly1*, which calls *ovly2*. *ovly2* just returns.

Here is the root:

```
main()
{
    ovloader("ovly1","in ovly1");
}
```

Here is ovly1:

```
ovmain(a)
char * a;
{
    printf("%s\n",a);
    ovloader("ovly2", "in ovly2");
}
```

Here is ovly2:

```
ovmain(a)
char *a;
{
    printf("%s\n",a);
}
```

The following commands link the root and the two overlays:

```
ln65 -R root.r ovloader.r -lc
ln65 -R ovly1.r ovbgn.r root.rsm -lc
ln65 ovly2.r ovbgn.r ovly1.rsm -lc
```

When executed, the following messages appear on the console:

```
in ovly1
in ovly2
```

2.5 Determining the size of the overlay area

When you link the root module, you will have to know how much memory to reserve for the overlay, that is, you will have to know how large the overlay is. But since the overlays haven't been linked yet, how can you know how much space is needed for overlays?

The easiest way is to guess. That is, estimate the size and go ahead and link the root and the overlays, keeping track of the size of the code and data for the overlays as reported by the linker.

After all overlays have been linked, the size of the area needed for overlays is the size of the largest overlay (if overlays aren't nested) or the size of the longest 'thread' of overlays (if they are nested). You can then go back and relink the root, if necessary, with this value. You won't have to relink any overlays, since the +C and +D options don't affect the position of the overlays in memory.

2.6 Error messages from ovloader

If an error occurs while loading an overlay, *ovloader* will print a message of the form

Error %d loading overlay: %s

where %d is a number defining the error and %s is the name of the overlay. The error codes and their meanings are:

10	Can't open overlay file
20	Can't read overlay header record
30	Invalid header record
40	Overlay code & data overlaps with heap
50	Error reading overlay

2.7 Possible Problems

A possible source of difficulty in using overlays concerns initialized data. In the following program module, a global variable is initialized:

```
int i = 3;
function()
{
    return;
}
```

The initialization of "i" is performed by the linker, rather than at run time. In the same program, the following module is allowed:

```
int i;
main()
{
    function();
}
```

The global variables in each module refer to the same integer, "i". At link time, this variable is set to the value 3. Although this works when the two modules are linked together, a problem arises when the first module is linked as an overlay:

```
ln65 func.r ovbgn.r main.rsm -lc
```

From the *.rsm* file, the linker knows that "int i" has been declared in *main.r*, the root. But it tries to initialize "i" from the statement in the *func.r* module. This attempt fails because the variable "i" is part of *main.r*, a module which is not included in the linkage.

An attempt to initialize, in an overlay, a variable which has been declared in the root will produce an error:

attempt to initialize data in root

The simple solution is to change the statement, "int i = 3", to the following:

```
int i;  
i = 3;
```

This assignment will be performed at run time, so that the linker does not try to perform an initialization.

2.8 Source

The source for the *ovloader* and *ovbgn* functions are in the files *ovld.c* and *ovbgn.a65*. *ovld* must be compiled by *cg65*; as mentioned above, it can be compiled with or without the option *-DPATH*, as defined above. *ovbgn* must be assembled using *as65*.

3. Interfacing to Assembly Language

This section discusses assembly-language functions that can call, or be called by C-language functions.

3.1 Naming Convention

The compilers translate a global function or variable name into assembler by truncating it to contain no more than 31 characters, appending an underscore character '_' to the truncated name, and then generating a *public* directive for the resultant name.

For example, the following assembly language statements define the entry point to an assembly language function that would be referred to in a C language program using the name *sum*:

```
public sum__  
sum__ ;entry point to sum
```

3.2 Calling and Returning

On entry to a function, information about the call are at the top of both the 6502 hardware stack and the pseudo stack.

At the top of the 6502 stack is the function's primary return address; this is the address to which the function should return by issuing an *rts* instruction. A non-reentrant function (ie, a function that doesn't call itself) can leave its return address on the 6502 stack and then return by issuing the 6502 *rts* instruction. For example, the very simplest assembly language function, which does nothing but return to the caller, would consist of just an *rts* instruction:

```
public nop__  
nop__ rts
```

Because of limitations of the 6502 stack, a reentrant function should save its return address on the pseudo stack. When done, it should return by doing an indirect *jmp* to the location whose address is one greater than the saved address.

3.3 Returning a value

A function can return an *int* or *long* value by setting the value in pseudo register R0, which is located in memory page 0. (The *equ* statements that defines R0 and all the other 0 page locations used by Aztec C-generated programs are in the file *zpage.h*). The bytes of the value are stored in order, with the least significant byte at address 8 and the most significant byte at the highest addressed location.

For example, here's a function that always returns the *int* value 1:

```
instxt  "zpage.h"
public  one_
one_    lda    #1
        sta    R0
        lda    #0
        sta    R0+1
        rts
```

3.4 Passing parameters

On entry to a function, the parameters that are being passed to the function and a secondary return address are on the pseudo stack, and are accessed using the field named SP that is located in memory page 0 and that points to the top of the pseudo stack. Note: as with R0, the *equ* statement that defines SP is in the file *zpage.h*.

At the top of the pseudo stack is the two-byte secondary return address. This is a different address from the return address that is on the 6502 stack - a function should return using the address that's on the 6502 stack. The secondary return address is discussed in the section of the Tech Info chapter that discusses the pseudo stack.

Above the secondary return address on the pseudo stack are the parameters that are being passed to the function. The function parameters are in order on the pseudo stack, with the first parameter immediately following the secondary return address, the second parameter following the first, and so on. The bytes for a parameter are also on the pseudo stack in order, with a parameter's least significant byte at the lowest address and its most significant byte at the highest address.

For example, suppose the function *sum* is passed two parameters, as follows:

```
sum(arg1, arg2);
```

On entry to *sum_*, the pseudo stack will look like this (SRA means "secondary return address"):

.....	
arg2, high byte	
arg2, low byte	
arg1, high byte	
arg1, low byte	
SRA, high byte	
SRA, low byte	<-- SP

3.5 An Example

The following assembly language function, named *sum*, is passed two *ints* as arguments. It returns their sum as its value.

```

instxt  "zpage.h"
public  sum__
sum__   clc
        ldy    #2
        lda    (SP),Y
        ldy    #4
        adc    (SP),Y
        sta    R0
        ldy    #3
        lda    (SP),Y
        ldy    #5
        adc    (SP),Y
        sta    R0+1
        rts

```

3.6 Page 0 Usage

A 6502 program makes extensive use of memory page 0. An assembly language 6502 function should obey the following restrictions on its usage of memory page 0 locations:

- * It may use, without preserving, the two-byte-long VAL field and the following four-byte-long fields: VAL, R0, R1, R2, R3, R4, and TMP.
- * It must preserve the contents of the SP, FRAME, and LFRAME (alias PC) fields and of the 16-byte REGS field.

These locations are defined in the file *zpage.h*.

3.7 Writing Programs that contain only Assembler

There are several topics concerning the linker which are important if the assembler and linker are to be used without any compiled code. The linker automatically creates several symbols that can be of use to an assembly language program, defining the beginning and end of the various program segments. These are described in the *Memory Organization* section of this chapter.

The entry point to a program is defined using the assembly language statement

```
entry loc
```

where *loc* is the name of the symbol where program execution is to begin. If a module containing an *entry* statement isn't encountered by the linker, it will set the program's entry point to the beginning of its code segment. For a discussion of the startup routines that are provided with Aztec C65, see the *Command Programs* section of this chapter.

3.8 Mixing C and Assembler in one Module

To include assembly language source in a C language module, surround the assembly language code with *#asm* and *#endasm* directives.

Finding a good example where this construct is necessary is very difficult, but here's a possible example:

```
rotate(arg)
{
    register int i;

    i = arg;
    #asm
        lda $81
        rol A
        rol $80
        rol $$1
    #endasm
    return(i);
}
```

This routine rotates a two byte quantity one bit to the left. This operation is messy in C and in a time critical application not feasible to make an assembly language subroutine. This routine is not a good example, since it would be better to write the entire thing in assembly. However, in the middle of a larger routine, it might conceivably be useful. This facility is provided as a last resort and is generally not recommended as it is completely non-portable.

4. Object module format

This section describes the format of object modules and libraries. The symbols and structures referred to in this paper are defined in the header file *object.h*.

4.1 Object Module Format

An object module contains four sections: header, code, table of named symbols, and table of unnamed symbols. These sections are described in the following paragraphs.

4.1.1 The Header Section

The header section of an object module has the following structure:

```
struct module {
    int          m__magic; /* type of object module */
    char         m__name[8]; /* module name */
    unsigned short m__code; /* module's code size */
    unsigned short m__data; /* module's data size */
    unsigned short m__static; /* module's bss data size */
    unsigned short m__global; /* named sym tbl off. */
    short         m__nglobal; /* # of named symbols */
    unsigned short m__local; /* unnamed sym tbl off. */
    short         m__nlocal; /* # of unnamed symbols */
    unsigned short m__end; /* unnamed sym tbl end */
    unsigned short m__next; /* offset to next module */
    unsigned short m__nfix /* # segment fixes required */
};
```

The following paragraphs discuss the fields within the *header* structure.

m__magic

Each of the different object module-related files created by the Aztec C software begins with the *m__magic* field, which contains a "signature" that identifies the file's contents. *m__magic* can have the following values:

M__MAGIC	Object module created by the assembler
M__OVROOT	Rsm file created by the linker
M__LIBRARY	Library of object modules

m__name

Contains the name of the object module. For object modules created by the assembler and for rsm files, this field normally contains null characters.

m__code, m__data, and m__static

Contain the size, in bytes, of an object module's code, data and uninitialized data segments, respectively.

m_global and m_nglobal

m_global contains the offset, in bytes, from the beginning of the module to the module's table of named symbols.
m_nglobal contains the number of entries in this table.

m_local and m_nlocal

m_local contains the offset, in bytes, from the beginning of the module to the module's table of unnamed symbols.
m_nlocal contains the number of entries in this table.

m_end

m_end contains the offset, in bytes, from the beginning of the module to the end of its table of unnamed symbols.

m_next

m_next contains the offset, in bytes, from the beginning of the module to the end of the module.

4.1.2 Symbol Tables

An object module contains two types of symbols: unnamed and named. An 'unnamed symbol' is a symbol whose name begins with a period followed by a digit. A 'named symbol' is any symbol that is not unnamed.

An object module has two symbol tables, one containing its named symbols, and the other its unnamed symbols. A symbol table contains entries, each of which describes one of the module's symbols. The entry for a symbol has the following structure:

```
struct symtab {
    char          s_type;    /* type of symbol */
    char          s_flags;   /* attributes of symbol */
    unsigned short s_value;  /* another attr of symbol */
}
```

In addition, the entry for a named symbol is followed by a null-terminated string, which is the symbol's name.

The following paragraphs discuss the fields of the *symtab* structure.

s_type

The *s_type* field in a symbol's table entry defines the type of the symbol. Possible values:

S_ABS	Symbol was defined to be a constant value, using the assembler's <i>equ</i> directive.
S_CODE	Symbol was defined within the code segment.
S_DATA	Symbol was defined within the data

	segment.
S__UND	Symbol was used but not defined within the program. Symbols that are defined using the assembler's <i>public</i> directive but aren't defined in any statement's label field have this type, as do symbols defined using the assembler's <i>global</i> directive. The directive used to define a S__UND symbol can be determined from the symbol's <i>s_value</i> field, as defined below.
S__BSS	Symbol was defined using the assembler's <i>bss</i> directive.

s_flags

This field defines other attributes of a symbol. Possible values:

S__GLOBL	Set for symbols specified in <i>public</i> and <i>global</i> directives.
S__FIXED	Set for symbols defined in some statement's label field.

s_value

The meaning of this field depends on the type of the symbol. Symbol types and their associated values are:

<i>s_type</i>	<i>Meaning of s_value</i>
S__ABS	Value specified for the symbol in the <i>equ</i> directive.
S__CODE	Offset of the symbol from the beginning of the module's code segment.
S__DATA	Offset of the symbol from the beginning of the module's data segment.
S__BSS	Size, in bytes, of the symbol as defined in the <i>bss</i> directive.
S__UND	For an S__UND symbol, <i>s_value</i> is zero if the symbol was defined in a <i>public</i> directive and non-zero if it was defined in a <i>global</i> directive. For a <i>global</i> -defined symbol, <i>s_value</i> contains the value specified in the directive's <i>size</i> operand.

4.1.3 The Code Section

The code section of an object module contains a translated version of the program. This format can be efficiently processed by the linker as it generates an executable version of the program. It contains a sequence of items, each of which directs the action of the linker. For example, some items contain actual code and data, which the linker

places in the output file, some cause the linker to reserve space in the output file, and some just pass information to the linker.

The linker builds several segments of a program simultaneously: a code segment, data segment, and an uninitialized data segment. Exactly one of these segments is said to be 'selected' at a time. There are loader items that select a segment.

The linker maintains a location counter for each of the segments that it is building. When a loader item requests that information be placed in the program or that space be reserved in it, the linker performs the requested operation in the current location of the currently-selected segment.

A loader item is a sequence of one or more bytes, with the first byte containing a code that identifies the item. Some codes are four bits long, and some are eight bits long; in the former case, the code occupies the most significant four bits of the byte.

Frequently, a loader item is two bytes long, with the item's code in the high order four bits of the item's first byte and a value in the other 12 bits. In this case, the value's least significant four bits are stored in the first byte's least significant four bits, and the value's most significant eight bits are stored in the second byte. We call this format "12-bit packed".

Descriptions of the loader items follow.

USECODE - Select code segment

The USECODE loader item selects the code segment. Data generated by loader items that follow the USECODE item will be placed in the code segment until another segment is selected.

The code for a USECODE loader item is 8 bits long: 0xf4.

USEDATA - Select initialed data segment

The USEDATA loader item selects the initialized data segment. Data generated by loader items that follow the USEDATA item will be placed in the code segment until another segment is selected.

The code for the USEDATA loader item is 0xf5.

ABSDAT - Absolute data

The *ABSDAT* loader item defines a sequence of bytes that the linker is to output 'as is' to the current location in the currently-selected segment.

The loader item's first byte contains the code identifier, 1, in the most significant four bits, and the number of bytes to be output, less one, in the least significant four bits. Thus,

this item can define one to sixteen bytes of absolute data. The remaining bytes in the item are the absolute data.

For example, the following *ABSDATA* loader item defines the three bytes A1, B2, and C3:

12 A1 B2 C3

LCLSYM - local (ie, unnamed) symbol

The value of a LCLSYM loader item is the address at which an unnamed symbol is located in memory.

The item is two bytes long, with the item's code, 6, in the first byte's most significant four bits. The item's other twelve bits contain the number of the symbol's entry in the local symbol table, in 12-bit packed format.

For example, given the assembly language code

dw	.98
.98 dw	12

with .98 occupying the second entry in the table of unnamed symbols, the following code would be generated for the *dw .98*:

61 00

GBLSYM - Global Symbol

The GBLSYM loader item is just like LCLSYM except that it references an entry in the global symbol table rather than the local symbol table.

The code for GBLSYM is the four-bit value 7.

SPACE - Reserve space

The SPACE loader item reserves a specified amount of space at the current location in the currently-selected segment.

The item is two bytes long, with the item's code, 8, in the most significant four bits of the item's first byte. The other twelve bits contain the number of bytes to reserve, less one, in 12-bit packed format.

For example, the following loader item reserves 5 bytes:

84 00

CODEREF - Code segment offset

The CODEREF loader item defines an offset from the beginning of the module's code segment. The loader item has as its value the absolute address corresponding to that offset.

The CODEREF loader item is in two bytes, with the CODEREF code, 0xa, in the high-order four bits of the item's first byte. The item's other 12 bits contain the offset, as a positive number, in 12-bit packed format.

DATAREF - Data segment offset

The DATAREF loader item is the same as the CODEREF loader item, except that the offset is relative to the beginning of the module's data segment.

The code for DATAREF is 0xb.

BSSREF - BSS segment offset

The BSSREF loader item is the same as the CODEREF loader item, except that the offset is relative to the beginning of the module's bss segment.

The code for BSSREF is 0xc.

LRGCODE - Code segment offset, large form

The LRGCODE loader item takes a 16-bit value that represents an offset from the beginning of its code segment, and generates as its value the absolute memory address of the location.

The loader item is in three bytes. The first byte contains the item's 8-bit code, 0xf7, the second contains the offset's least significant eight bits, and the third contains the offset's most significant eight bits.

LRGDATA - Data segment offset, large form

The LRGDATA loader item is the same as LRGCODE except that the offset is relative to the beginning of the module's data segment.

The code for the LRGDATA loader item is 0xf8.

LRGBSS - BSS segment offset, large form

The LRGBSS loader item is the same as LRGCODE except that the offset is relative to the beginning of the module's BSS segment.

The code for the LRGBSS loader item is 0xfb.

SMLINT - small integer

The SMLINT loader item defines an integer between 0 and 15, inclusive. This item can be used by itself or as an element of an EXPR loader item.

The loader item consists of a single byte. Its most significant four bits are the item's code, 3; and the least

significant four bits are the integer value.

For example, the following defines the integer value 8:

38

SMLNEG - Small negative integer

The SMLNEG loader item defines a negative integer between -1 and -16 inclusive. It can be used by itself or in an EXPR loader item.

The loader item is a single byte: the high order 4 bits are the item's code, 4. The low order four bits are the absolute value of the integer, less 1.

For example, the following defines the negative value -8:

47

MEDINT

The MEDINT loader item defines an integer in the range -2048 to 2047, inclusive, that can be used by itself or in an EXPR loader item.

The item consists of two bytes, with the high-order four bits of the least significant byte containing the item's code, 5, and the remaining twelve bits defining the value, in 12-bit packed format.

The value is in 'excess-2048' notation. The number actually in the 12-bit field is an integer between 0 and 4095; the integer denoted by the item is derived from the actual integer by subtracting 2048 from it.

For example, the following represents the value -1024:

50 40

LRGINT - Large integer

The LRGINT loader item defines an integer in the range -32K to +32K, for use in an expression loader item.

The item consists of three bytes. Its first byte contains the 8-bit code identifying the item, 0xf3. The other two bytes contain the value, in two's-complement notation.

EXPR - Evaluate expression

The EXPR loader item has as its value the 16-bit value of the expression that follows it. The size of the loader item depends on the size of the items that comprise the expression. The most significant four bits of the item's first byte contains the code for the loader item, 2, and the least significant four bits contain a code for the operation that is to be performed

on the loader items that follow. The codes and their corresponding values and operations are:

<i>code</i>	<i>value</i>	<i>operation</i>
ADD	1	Add the two loader items that follow
SUB	2	Subtract the following two loader items
MUL	3	Multiply the following two loader items
DIV	4	Divide the first item that follows by the second
MOD	5	Compute the modulus of the first item relative to the second.
AND	6	Logical AND of the following two items
OR	7	Logical OR of the following two items
XOR	8	Exclusive OR of the following two items
RSH	9	Right shift first item the number of bits defined by second item
LSH	10	Left shift first item the number of bits defined by the second
NOT	11	Logical NOT of item that follows
NEG	12	Compute two's complement of the item that follows

The items that can follow an EXPR item are SMLINT, MEDINT, LRGIN, LCLSYM, GBLSYM, CODEREF, DATAREF, BSSREF, LRDCODE, LRDDATA, LRDBSS, and another EXPR.

For example, given the assembly language code

dw a+4

with the entry for *a* being the fourth entry in the table of named symbols, the following loader items would be generated:

21 73 00 34

As mentioned above, an EXPR can have another EXPR as one of its loader items. In this case, the inner EXPR is evaluated, using the loader items that follow it, and then the outer EXPR is evaluated, using the resultant value of the inner EXPR as one value and whatever loader items are left for the other values. The loader items for the entire expression are thus in prefix-Polish notation. For example, the above expression, *a+4*, is represented by the loader items that correspond to

+a4

And the expression

$(a+b)*c$

would be represented by loader items that correspond to

$*+abc$

BEXPR - Evaluate byte expression

The BEXPR loader item has as its value the 8-bit value of the expression that follows it. BEXPR has an 8-bit code, 0xf1. BEXPR doesn't have an extra four bits in which an operation code can be placed; thus, to generate an 8-bit value from an expression, a BEXPR loader item will usually precede an EXPR loader item that is in turn followed by the loader items for the expression.

BREL - compute offset from location counter, byte form

The BREL loader item takes a relocatable value that represents a location in the module and generates the offset of the location from the current location counter.

The BREL loader item begins with a 8-bit code, 0xf2. It's followed by loader items representing the location.

For example, if the symbol *abc* is the fourth symbol in the global symbol table, then the loader items to generate the offset of the location that is four bytes beyond *abc* are

f2 21 73 00 34

WREL - compute offset from location counter, word form

The WREL loader item is the same as BREL except that it generates a 16-bit value instead of an 8-bit value.

STARTAD - Define program start address

The STARTAD loader item defines the address at which a program containing the module is to begin execution.

The item begins with the item's 8-bit code, 0xf6. It's followed by loader items identifying the starting address; these can be GBSYM, LCLSYM, EXPR, or any of the other "expression items" mentioned above.

INTJSR - Generate opcode for a subroutine call

The INTJSR loader item is translated by the linker into a machine-specific opcode that will cause a subroutine to be called. The loader item has the value 0xf9.

The instructions in a function that has been compiled with the interpretive compiler consist of a call to the Aztec interpreter routine followed by the function's other instructions. This first instruction is directly executed by the

machine; the function's other instructions are in a pseudo code that is indirectly executed, by the Aztec interpreter.

It is desirable to allow the interpretive compiler to generate object modules that can be executed on different machines, and to allow a single object module generated using this compiler to be linked for execution on different processor chips. To support this, the interpretive compiler generates as a function's first instruction a special call instruction, in the pseudo code assembly language, to the interpreter. The pseudo code assembler translates this instruction into an INTJSR loader item followed by a GBSYM loader item that references the interpreter routine. The machine-specific linker then translates this pair of loader items into a machine-specific call to the interpreter.

THEEND - End of code

The THEEND loader item identifies the end of the code section of the object file.

The code for the item is 00.

4.2 Object Library Format

A library of object modules consists of the object modules and a directory of symbol names.

4.2.1 Object Modules in a Library

When an object module is placed in a library its sections are reorganized but the contents of the module are left unchanged (with the exception of the module's header, whose fields are modified to reflect the reorganization). The module's header still is at the beginning of the module. This is followed by the table of named symbols, the table of unnamed symbols, and the code section.

The header is modified to define the positions of the tables in the reorganized module, and the module is given a name in its *m_name* field. The name is derived from the name of the file that contained the module by removing the file name's extension.

4.2.2 Library Dictionary

A library's dictionary consists of one or more blocks that are chained together. A block has the following structure:

```
struct newlib {
    short nl_magic;           /* magic number for libraries */
    unsigned short nl_next;   /* loc of next dir block */
    char nl_dict[LBSIZE];     /* dictionary for block */
}
```

nl_dict contains entries, each of which defines one symbol that is defined in a library module. The entry for a symbol consists of a short int that defines the position of the module that defines the symbol (the absolute location at which the module begins, divided by 128), and a null-terminated string that is the symbol's name.

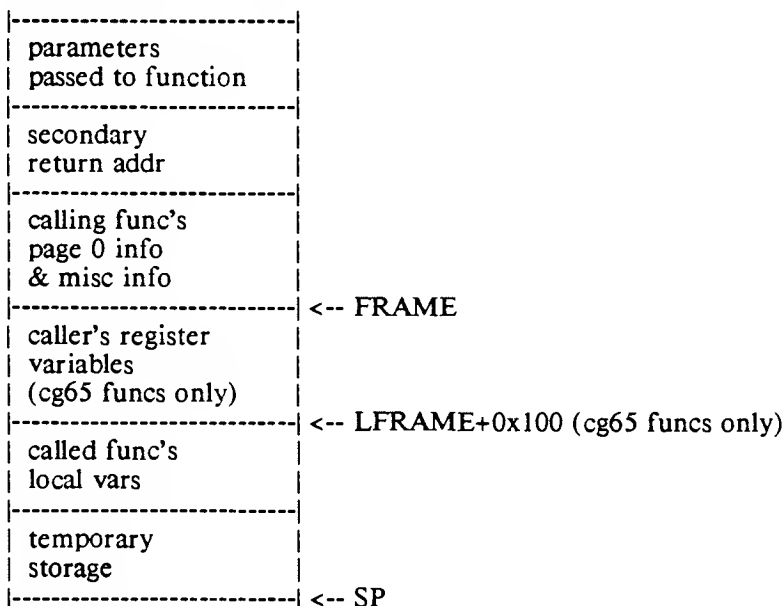
5. The pseudo stack

Information in the zero page and in the pseudo stack can be used in conjunction with a linker-generated symbol table to help debug a program. For example, when a program mysteriously aborts and exits to the monitor, this information can be used to determine where the program was and how it got there.

During the execution of a program, the pseudo stack contains a list of "frames", each of which contains information about a function that has been called but hasn't returned. A function's frame defines the parameters that were passed to it, the address to which it will return, the values of its local variables, information about the function that called it, and other information.

At the top of the pseudo stack is the frame for the "active" function; that is, about the currently-executing function. Above that is the frame for the function that called the active function; above that is the frame for the function that called the function that called the active function, and so on, back to the frame for the first function called by the program's startup code.

A function's frame has the following organization:



In the above diagram, SP, FRAME, and LFRAME are the names of zero-page fields that point to areas within the frame of the active function. These fields are defined in the file *zpage.h*, along with other zero page fields used by Aztec C-compiled functions, as described in the *Memory Organization* section of the *Tech Info* chapter.

The LFRAME field is used for two purposes: when a function that has been compiled with the *cg65* compiler is active, this field goes by the name LFRAME and points into the active function's frame. When a *cci*-compiled function is active, this field goes by the name PC and acts as a program counter, pointing to the next pseudo-code instruction that is to be executed by the Aztec interpreter routine.

Locations in the active function's frame are specified by adding a value to the contents of a zero page field. To abbreviate the definition of these locations, the following paragraphs will refer to them using an expression consisting of the parenthesized name of the zero-page field plus or minus the value. For example, the expression (FRAME)+11 refers to the location within the active function's frame whose address is obtained by adding eleven to the contents of the zero page field named FRAME.

5.1 Secondary Return Address

The secondary return address field in a called function's frame, which we'll refer to here as SRA, defines the address within the calling function at which execution will continue when the called function returns.

To be exact, if the calling function was compiled with *cg65*, execution within it will continue at the address (SRA)+1; ie, at the location whose address is one greater than that contained in the called function's secondary return address field.

If the calling function was compiled with *cci* and if no parameters were passed to the called function, execution of pseudo-code instructions within the calling function by the interpreter will resume at address (SRA). If parameters were passed, execution will instead resume at address (SRA)+1.

The secondary return address field for the active function is in the two-byte field the begins at address (FRAME)+9; ie, 9 bytes above the location within the active function's frame that is pointed at by the zero page FRAME field.

5.2 Determining the function in which a program aborted

When a program aborts and exits to the monitor, the first thing you should do is determine the identity of the active function. This can be done as follows:

1. Find the active function's secondary return address;
2. In the code that precedes this address, find the address of the active function;
3. From the program's linker-generated symbol table, find the name of the active function.

If the address of the active function isn't in this table, because the function is declared to be *static*, you can at least determine from an

examination of the symbol table the module in which it was defined.

The function calling sequences are different for *cg65*- and *cci*-compiled functions. So the following paragraphs first describe the code generated for a function call by the two compilers and then describe how to examine it to find the address of a called function.

5.2.1 Calling sequence for *cg65*-compiled functions

The *cg65* compiler translates a direct function call into 6502 code that first pushes the arguments onto the 6502 stack and then issues a *jsr* to the Aztec routine *.cpystk*. Following the *jsr* is a two-byte field that contains the address of the called function and then a one-byte field that defines the number of bytes that the called function's parameters and secondary return address will occupy on the pseudo stack. The secondary return address of the called function is set to the calling sequence's one-byte field.

For example, suppose the following call is made to the function *func*:

```
func(a,b,c,d)
```

The compiler will first generate code to push *d, c, b, a* (in that order) onto the 6502 stack. Then it will generate the following code:

```
jsr    .cpystk
fdw    func__
fcb    10
```

.cpystk will pull the arguments off the hardware stack, push them onto the pseudo stack, push the address of the *fcb 10* onto the pseudo stack and issue a *jsr* to *func*. The address of the *fcb 10* is the called function's secondary return address.

cg65 translates an indirect function call (eg, *(*foo)()*) into 6502 code that pushes the arguments on the 6502 stack, moves the address of the function into R0 (the zero-page simulated register), and issues a *jsr* to the Aztec routine *.cpystk2*. Then *cg65* generates a one-byte field that defines the number of bytes on the pseudo stack that the function's parameters and secondary return address will use. The secondary return address of the called function is set to the address of the one-byte field.

5.2.2 Calling sequence for *cci*-compiled functions

The *cci* compiler translates a direct function call by first generating pseudo-code that pushes the parameters onto the pseudo stack. It then generates a three-byte call pseudo-instruction, consisting of an op code (0xac if no parameters are specified in the call, 0xe9 if there are parameters) and a two byte field containing the address of the called function. The secondary return address of the called function is set to the byte that follows the interpreter's call instruction.

cci translates an indirect function call into pseudo code that first pushes the parameters onto the pseudo stack, then loads the called function's address into R0. It then generates a one-byte call pseudo instruction (0xdd if no parameters are specified, 0xea if they are). The secondary return address of the called function is set to the address of the byte following the call instruction.

5.2.3 Examining the calling sequence

To find the address of the active function from the sequence of instructions that called it, you should examine the bytes that precede the function's secondary return address:

- * If the fifth through the third preceding bytes are *jsr .cpystk* (indicating a direct function call from a *cg65*-compiled function) or if the third preceding byte is 0xdd or 0xea (a direct function call from a *cci*-compiled function), the second and first preceding bytes contain the address of the function.
- * If the fifth through the third preceding bytes are *jsr .cpystk2* (an indirect function call from a *cg65*-compiled function), or if the third preceding byte is 0xac or 0xe9 (an indirect function call from a *cci*-compiled function), you'll have to find the function address by examining the variables from which the function address was computed.

5.3 Determining the parameters passed to the active function

To determine the parameters that have been passed to the active function, you should first determine the identity of the active function. This knowledge will then give you the number and types of the function's parameters. You can then simply examine the function's arguments on the pseudo stack: the first parameter begins at address (FRAME)+11 and occupies the number of bytes appropriate for a value of its type. The second parameter begins immediately above the first, and occupies the required number of bytes, and so on.

5.4 Determining the values of the active function's local variables

The active function's local variables occupy a section of the function's frame on the pseudo stack. This section extends downward from the first byte below the location pointed at either (1) by the zero-page LFRAME field, if the function was compiled by *cg65* or (2) by the zero-page FRAME field, if it was compiled by *cci*.

Local variables are allocated space in a function's frame in the order in which they are defined, at successively decreasing locations. For example, consider the following function:

```

foo()
{
    int a,b,c;
    ...
}

```

The local variable *a* will occupy the first two bytes below the location pointed at by LFRAME (for a *cg65*-compiled function) or FRAME (for a *cci*-compiled function); *b* will occupy the next two bytes, and *c* will occupy the next two bytes.

5.5 Determining the values of register variables for the active function

Register variables are supported only for *cg65*-compiled functions. There are eight two-byte pseudo registers. They are in the zero page, beginning at the location whose name is REG (defined in *zpage.h* to be 0x80).

Variables are allocated to registers in the order in which their declarations are encountered. For example, consider the following function:

```

foo(a,b,c)
register int b;
{
    int d;
    register e;
    int f;
    ...
}

```

The variable *b* will occupy the register at addresses REG and REG+1, and the variable *d* occupies the register at REG+2 and REG+3.

5.6 Function entry and exit

When a function is entered, the zero page fields SP, FRAME, and LFRAME are saved, and updated for the new function. The saved values are then moved into the new function's frame, in locations (FRAME)+2, (FRAME)+4, and (FRAME)+6. When the function is exited, these fields are restored.

When a function is entered, its primary return address, which is on the top of the hardware stack, is saved in the new function's frame, in location (FRAME).

When a C function calls another function, the call is indirectly made by transferring control to an intermediary routine, which in turn calls the other function. When the called function returns, control is again transferred to the intermediary routine, which then returns to the calling function. A called function's primary return address is the address in the intermediary routine to which control is returned by issuing an *rts* from the called function. And the called function's

secondary return address is the address of the calling function to which the intermediary routine returns. On entry to a called function, its primary return address is at the top of the 6502 hardware stack and its secondary return address is at the top of the pseudo stack.

5.7 Getting information about a calling function

Once you've gotten all the information you can about the active function, using its frame on the pseudo stack, you can get information about the function that called it by examining the calling function's frame on the pseudo stack. If necessary, you can continue examining frames on the pseudo stack until you know the state of all the function's that have been called but that have not yet returned. In the following discussion, we'll call the function that called the active function function 2, the function that called it function 3, and so on.

First of all, since the active function's secondary return address, whose value you know, is the address of the location in function 2 (ie, the calling function) to which the active function will return, you can scan the program's symbol table and learn the identity of function 2.

In the active function's frame, the two-byte fields at (FRAME)+4 and (FRAME)+6 contain the values that were in the FRAME and LFRAME fields at the time function 2 was active. Using these values, you can examine function 2's frame and determine the parameters that were passed to it and the values of its local variables. You can also determine the identity of function 3 (ie, the function that called function 2) from the secondary return address field within function 2's frame, and you can locate function 3's frame using the fields in function 2's frame that were in the FRAME and LFRAME fields when function 2 was active.

OVERVIEW OF LIBRARY FUNCTIONS

Chapter Contents

Overview of Library Functions libov

1. I/O Overview 4

1.1 Pre-opened devices, command line args 4

1.2 File I/O 6

1.2.1 Sequential I/O 6

1.2.2 Random I/O 6

1.2.3 Opening Files 6

1.3 Device I/O 7

1.3.1 Console I/O 7

1.3.2 I/O to Other Devices 7

1.4 Mixing unbuffered and standard I/O calls 7

2. Standard I/O Overview 9

2.1 Opening files and devices 9

2.2 Closing Streams 9

2.3 Sequential I/O 10

2.4 Random I/O 10

2.5 Buffering 10

2.6 Errors 11

2.7 The standard I/O functions 12

3. Unbuffered I/O Overview 14

3.1 File I/O 15

3.2 Device I/O 15

3.2.1 Unbuffered I/O to the Console 15

3.2.2 Unbuffered I/O to Non-Console Devices 16

4. Console I/O Overview 17

4.1 Line-oriented input 17

4.2 Character-oriented input 18

4.3 Using ioctl 19

4.4 The sgty fields 19

4.5 Examples 20

5. Dynamic Buffer Allocation 22

6. Error Processing Overview 23

Overview of Library Functions

This chapter presents an overview of the functions that are provided with Aztec C. It's divided into the following sections:

1. *I/O*: Introduces the i/o system provided in the Aztec C package.
2. *Standard I/O*: The i/o functions can be grouped into two sets; this section describes one of them, the standard i/o functions.
3. *Unbuffered I/O*: Describes the other set of i/o functions, the unbuffered.
4. *Console I/O*: Describes special topics relating to console i/o.
5. *Dynamic Buffer Allocation*: Discusses topics related to dynamic memory allocation.
6. *Errors*: Presents an overview of error processing.

The overviews present information that is system independent. Overview information that is specific to your system is in the form of an appendix to this chapter; it accompanies the system dependent section of your manual.

1. Overview of I/O

There are two sets of functions for accessing files and devices: the unbuffered i/o functions and the standard i/o functions. These functions are identical to their UNIX equivalents, and are described in chapters 7 and 8 of *The C Programming Language*.

The unbuffered i/o functions are so called because, with few exceptions, they transfer information directly between a program and a file or device. By contrast, the standard i/o functions maintain buffers through which data must pass on its journey between a program and a disk file.

The unbuffered i/o functions are used by programs which perform their own blocking and deblocking of disk files. The standard i/o functions are used by programs which need to access files but don't want to be bothered with the details of blocking and deblocking the file records.

The unbuffered and standard i/o functions each have their own overview section (UNBUFFERED I/O and STANDARD I/O). The remainder of this section discusses features which the two sets of functions have in common.

The basic procedure for accessing files and devices is the same for both standard and unbuffered i/o: the device or file must first be "opened", that is, prepared for processing; then i/o operations occur; then the device or file is "closed".

There is a limit on the number of files and devices that can simultaneously be open; the limit on your system is defined in this chapter's system dependent appendix.

Each set of functions has its own functions for performing these operations. For example, each set has its own functions for opening a file or device. Once a file or device has been opened, it can be accessed only by functions in the same set as the function which performed the open, and must be closed by the appropriate function in the same set. There are exceptions to this non-intermingling which are described below.

There are two ways a file or device can be opened: first, the program can explicitly open it by issuing a function call. Second, it can be associated with one of the logical devices standard input, standard output, or standard error, and then opened when the program starts.

1.1 Pre-opened devices and command line arguments

There are three logical devices which are automatically opened when a program is started: standard input, standard output, and standard error. By default, these are associated with the console. The operator, as part of the command line which starts the program, can specify that these logical devices are to be "redirected" to another

device or file. Standard input is redirected by entering on the command line, after the program name, the name of the file or device, preceded by the character '<'. Standard output is redirected by entering the name of the file or device, preceded by '>'.

For example, suppose the executable program *cpy* reads standard input and writes it to standard output. Then the following command will read lines from the keyboard and write them to the display:

```
cpy
```

The following will read from the keyboard and write it to the file *testfile*:

```
cpy >testfile
```

This will copy the file *exmplfil* to the console:

```
cpy <exmplfil
```

And this will copy *exmplfil* to *testfile*:

```
cpy <exmplfil >testfile
```

Aztec C will pass command line arguments to the user's program via the user's function *main(argc, argv)*. *argc* is an integer containing the number of arguments plus one; *argv* is a pointer to an array of character pointers, each of which, except the first, points to a command line argument. On some systems, the first array element points to the command name; on others, it is a null pointer. Information on your system's treatment of this pointer is presented in this chapter's system dependent appendix.

For example, if the following command is entered:

```
prog arg1 arg2 arg3
```

the program *prog* will be activated and execution begins at the user's function *main*. The first parameter to *main* is the integer 4. The second parameter is a pointer to an array of four character pointers; on some systems the first array element will point to the string "prog" and on others it will be a null pointer. The second, third, and fourth array elements will be pointers to the strings "arg1", "arg2", and "arg3" respectively.

The command line can contain both arguments to be passed to the user's program and i/o redirection specifications. The i/o redirection strings won't be passed to the user's program, and can appear anywhere on the command line after the command name. For example, the standard output of the "prog" program can be redirected to the file *outfile* by any of the following commands; in each case the *argc* and *argv* parameters to the *main* function of 'prog' are the same as if the redirection specifier wasn't present:

```
prog arg1 arg2 arg3 >outfile  
prog >outfile arg1 arg2 arg3  
prog arg1 >outfile arg2 arg3
```

1.2 File I/O

A program can access files both sequentially and randomly, as discussed in the following paragraphs.

1.2.1 Sequential I/O

For sequential access, a program simply issues any of the various read or write calls. The transfer will begin at the file's "current position", and will leave the current position set to the byte following the last byte transferred. A file can be opened for read or write access; in this case, its current position is initially the first byte in the file. A file can also be opened for append access; in this case its current position is initially the end of the file.

On systems which don't keep track of the last character written to a file, it isn't always possible to correctly position a file to which data is to be appended. If this is a problem on your system, it's discussed in the system dependent appendix to this chapter, which accompanies the system dependent section of your manual.

1.2.2 Random I/O

Two functions are provided which allow a program to set the current position of an open file: *fseek*, for a file opened for standard i/o; and *lseek*, for a file opened for unbuffered i/o.

A program accesses a file randomly by first modifying the file's current position using one of the seek functions. Then the program issues any of the various read and write calls, which sequentially access the file.

A file can be positioned relative to its beginning, current position, or end. Positioning relative to the beginning and current position is always correctly done. For systems which don't keep track of the last character written to a file, positioning relative to the end of a file can't always be correctly done. For information on this, see this chapter's system dependent appendix.

1.2.3 Opening files

Opening files is somewhat system dependent: the parameters to the open functions are the same on the Aztec C packages for all systems, but some system dependencies exist, to conform with the system conventions. For example, the syntax of file names and the areas searched for files differ from system to system.

For information on the opening of files on your system, see this chapter's system dependent appendix.

1.3 Device I/O

Aztec C allows programs to access devices as well as files. Each system has its own names for devices: for the names of devices on your system, see this chapter's system dependent appendix.

1.3.1 Console I/O

Console I/O can be performed in a variety of ways. There's a default mode, and other modes can be selected by calling the function *ioctl*. We'll briefly describe console I/O in this section; for more details, see the *Console I/O* section of this chapter and the system dependent appendix to this chapter.

When the console is in default mode, console input is buffered and is read from the keyboard a line at a time. Typed characters are echoed to the screen and the operator can use the standard operating system line editing facilities. A program doesn't have to read an entire line at a time (although the system software does this when reading keyboard input into it's internal buffer), but at most one line will be returned to the program for a single read request.

The other modes of console i/o allow a program to get characters from the keyboard as they are typed, with or without their being echoed to the display; to disable normal system line editing facilities; and to terminate a read request if a key isn't depressed within a certain interval.

Output to the console is always unbuffered: characters go directly from a program to the display. The only choice concerns translation of the newline character; by default, this is translated into a carriage return, line feed sequence.

Optionally, this translation can be disabled.

1.3.2 I/O to Other Devices

On most systems, few options are available when writing to devices other than the console. For a discussion of such options, if any, that are available on your system, see this chapter's system dependent appendix.

1.4 Mixing unbuffered and standard i/o calls

As mentioned above, a program generally accesses a file or device using functions from one set of functions or the other, but not both.

However, there are functions which facilitate this dual access: if a file or device is opened for standard i/o, the function *fileno* returns a file descriptor which can be used for unbuffered access to the file or device. If a file or device is open for unbuffered i/o, the function *fdopen* will prepare it for standard i/o as well.

Care is warranted when accessing devices and files with both standard and unbuffered i/o functions.

2. Overview of Standard I/O

The standard i/o functions are used by programs to access files and devices. They are compatible with their UNIX counterparts, with few exceptions, and are also described in chapter 8 of *The C Programming Language*. The exceptions concern appending data to files and positioning files relative to their end, and are discussed below.

These functions provide programs with convenient and efficient access to files and devices. When accessing files, the functions buffer the file data; that is, handle the blocking and deblocking of file data. Thus the user's program can concentrate on its own concerns.

Buffering of data to devices when using the standard i/o functions is discussed below.

For programs which perform their own file buffering, another set of functions are provided. These are described in the section UNBUFFERED I/O.

2.1 Opening files and devices

Before a program can access a file or device, it must be "opened", and when processing on it is done it must be "closed".

An open device or file is called a "stream" and has associated with it a pointer, called a "file pointer", to a structure of type FILE. This identifies the file or device when standard i/o functions are called to access it.

There are two ways for a file or device to be opened for standard i/o: first, the program can explicitly open it, by calling one of the functions *fopen*, *freopen*, or *fdopen*. In this case, the open function returns the file pointer associated with the file or device. *fopen* just opens the file or device. *freopen* reopens an open stream to another file or device; it's mainly used to change the file or device associated with one of the logical devices standard output, standard input, or standard error. *fdopen* opens for standard i/o a file or device already opened for unbuffered i/o.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file pointer is *stdin*, *stdout*, or *stderr*, respectively. These symbols are defined in the header file *stdio.h*. See the section entitled I/O for more information on logical devices.

2.2 Closing streams

A file or device opened for standard i/o can be closed in two ways: first, the program can explicitly close it by calling the function *fclose*.

Alternatively, when the program terminates, either by falling off the end of the function *main*, or by calling the function *exit*, the system will automatically close all open streams.

Letting the system automatically close open streams is error-prone: data written to files using the standard i/o functions is buffered in memory, and a buffer isn't written to the file until it's full or the file is closed. Most likely, when a program finishes writing to a file, the file's buffer will be partially full, with this information not having been written to the file. If a program calls *fclose*, this function will write the partially filled buffer to the file and return an error code if this couldn't be done. If the program lets the system automatically close the file, the program won't know if an error occurred on this last write operation.

2.3 Sequential I/O

Files can be accessed sequentially and randomly. For sequential access, simply issue repeated read or write calls; each call transfers data beginning at the "current position" of the file, and updates the current position to the byte following the last byte transferred. When a file is opened, its current position is set to zero, if opened for read or write access, and to its end if opened for append.

On systems which don't keep track of the last character written to a file, such as CP/M and Apple // DOS, not all files can be correctly positioned for appending data. See the section entitled I/O for details.

2.4 Random I/O

The function *fseek* allows a file to be accessed randomly, by changing its current position. Positioning can be relative to the beginning, current position, or end of the file.

For systems which don't keep track of the last character written to a file, such as CP/M and Apple // DOS, positioning relative to the end of a file cannot always be correctly done. See the I/O overview section for details.

2.5 Buffering

When the standard i/o functions are used to access a file, the i/o is buffered. Either a user-specified or dynamically- allocated buffer can be used.

The user's program specifies a buffer to be used for a file by calling the function *setbuf* after the file has been opened but before the first i/o request to it has been made.

If, when the first i/o request is made to a file, the user hasn't specified the buffer to be used for the file, the system will automatically allocate, by calling *malloc*, a buffer for it. When the file is closed it's buffer will be freed, by calling *free*.

Dynamically allocated buffers are obtained from the one region of memory (the heap), whether requested by the standard i/o functions or by the user's program. For more information, see the overview

section *Dynamic Buffer Allocation*.

The size of an i/o buffer differs from system to system. See this chapter's system-dependent appendix for the size of this buffer on your system.

A program which both accesses files using standard i/o functions and has overlays has to take special steps to insure that an overlay won't be loaded over a buffer dynamically allocated for file i/o. For more information, see the section on overlay support in the *Technical Information* chapter.

By default, output to the console using standard i/o functions is unbuffered; all other device i/o using the standard i/o functions is buffered. Console input buffering can be disabled using the *ioctl* function; see the overview section *Console I/O* for details.

2.6 Errors

There are three fields which may be set when an exceptional condition occurs during stream i/o. Two of the fields are unique to each stream (that is, each stream has its own pair). The other is a global integer.

One of the fields associated with a stream is set if end of file is detected on input from the stream; the other is set if an error occurs during i/o to the stream. Once set for a stream, these flags remain set until the stream is closed or the program calls the *clearerr* function for the stream. The only exception to the last statement is that when called, *fseek* will reset the end of file flag for a stream. A program can check the status of the eof and error flags for a stream by calling the functions *feof* and *ferror*, respectively.

The other field which may be set is the global integer *errno*. By convention, a system function which returns an error status as its value can also set a code in *errno* which more fully defines the error. The overview section *Errors* defines the values which may be set in *errno*.

If an error occurs when a stream is being accessed, a standard i/o function returns EOF (-1) as its value, after setting a code in *errno* and setting the stream's error flag.

If end of file is reached on an input stream, a standard i/o function returns EOF after setting the stream's eof flag.

There are two techniques a program can use for detecting errors during stream i/o. First, the program can check the result of each i/o call. Second, the program can issue i/o calls and only periodically check for errors (for example, check only after all i/o is completed).

On input, a program will generally check the result of each operation.

On output to a file, a program can use either error checking technique; however, periodic checking by calling *ferror* is more efficient. When characters are written to a file using the standard i/o functions they are placed in a buffer, which is not written to disk until it is full. If the buffer isn't full, the function will return good status. It will only return bad status if the buffer was full and an error occurred while writing it to disk. Since the buffer size is 1024 bytes, most write calls will return good status, and hence periodic checking for errors is sufficient and most efficient.

Once a file opened for standard i/o is closed, *ferror* can't be used to determine if an error has occurred while writing to it. Hence *ferror* should be called after all writing to the file is completed but before the file is closed. The file should be explicitly closed by *fclose*, and its return value checked, rather than letting the system automatically close it, to know positively whether an error has occurred while writing to the file. The reason for this is that when the writing to the file is completed, it's standard i/o buffer will probably be partly full. This buffer will be written to the file when the file is closed, and *fclose* will return an error status if this final write operation fails.

2.7 The standard i/o functions

The standard i/o functions can be grouped into two sets: those that can access only the logical devices standard input, standard output, and standard error; and all the rest.

Here are the standard i/o functions that can only access *stdin*, *stdout*, and *stderr*. These are all ASCII functions; that is, they expect to deal with text characters only.

<code>getchar</code>	Get an ASCII character from <i>stdin</i>
<code>gets</code>	Get a line of ASCII characters from <i>stdin</i>
<code>printf</code>	Format data and send it to <i>stdout</i>
<code>puterr</code>	Send a character to <i>stderr</i>
<code>putchar</code>	Send a character to <i>stdout</i>
<code>puts</code>	Send a character string to <i>stdout</i>
<code>scanf</code>	Get a line from <i>stdin</i> and convert it

Here are the rest of the standard i/o functions:

agetc	Get an ASCII character
aputc	Send an ASCII character
fopen	Open a file or device
fdopen	Open as a stream a file or device already open for unbuffered i/o
freopen	Open an open stream to another file or device
fclose	Close an open stream
feof	Check for end of file on a stream
ferror	Check for error on a stream
fileno	Get file descriptor associated with stream
fflush	Write stream's buffer
fgetc	Get a line of ASCII characters
fprintf	Format data and write it to a stream
fputs	Send a string of ASCII characters to a stream
fread	Read binary data
fscanf	Get data and convert it
fseek	Set current position within a file
ftell	Get current position
fwrite	Write binary data
getc	Get a binary character
getw	Get two binary characters
putc	Send a binary character
putw	Send two binary characters
setbuf	Specify buffer for stream
ungetc	Push character back into stream

3. Overview of Unbuffered I/O

The unbuffered I/O functions are used to access files and devices. They are compatible with their UNIX counterparts and are also described in chapter 8 of *The C Programming Language*.

As their name implies, a program using these functions, with two exceptions, communicates directly with files and devices; data doesn't pass through system buffers. Some unbuffered I/O, however, is buffered: when data is transferred to or from a file in blocks smaller than a certain value, it is buffered temporarily. This value differs from system to system, but is always less than or equal to 512 bytes. Also, console input can be buffered, and is, unless specific actions are taken by the user's program.

Programs which use the unbuffered i/o functions to access files generally handle the blocking and deblocking of file data themselves. Programs requiring file access but unwilling to perform the blocking and deblocking can use the standard i/o functions; see the overview section *Standard I/O* for more information.

Here are the unbuffered i/o functions:

<code>open</code>	Prepares a file or device for unbuffered i/o
<code>creat</code>	Creates a file and opens it
<code>close</code>	Concludes the i/o on an open file or device
<code>read</code>	Read data from an open file or device
<code>write</code>	Write data to an open file or device
<code>lseek</code>	Change the current position of an open file
<code>rename</code>	Renames a file
<code>unlink</code>	Deletes a file
<code>ioctl</code>	Change console i/o mode
<code>isatty</code>	Is an open file or device the console?

Before a program can access a file or device, it must be "opened", and when processing on it is done, it must be "closed".

An open file or device has an integer known as a "file descriptor" associated with it; this identifies the file or device when it's accessed.

There are two ways for a file or device to be opened for unbuffered i/o. First, it can explicitly open it, by calling the function *open*. In this case, *open* returns the file descriptor to be used when accessing the file or device.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file descriptor is the integer value 0, 1, or 2, respectively. See the section entitled I/O for more information on this.

An open file or device is closed by calling the function *close*. When a program ends, any devices or files still opened for unbuffered i/o will be closed.

If an error occurs during an unbuffered i/o operation, the function returns -1 as its value and sets a code in the global integer *errno*. For more information on error handling, see the section ERRORS.

The remainder of this section discusses unbuffered i/o to files and devices.

3.1 File I/O

Programs call the functions *read* and *write* to access a file; the transfer begins at the "current position" of the file and proceeds until the number of characters specified by the program have been transferred.

The current position of a file can be manipulated in various ways by a program, allowing both sequential and random access to the file. For sequential access, a program simply issues consecutive i/o requests. After each operation, the current position of the file is set to the character following the last one accessed.

The function *lseek* provides random access to a file by setting the current position to a specified character location.

lseek allows the current position of a file to be set relative to the end of a file. For systems which don't keep track of the last character written to a file, such positioning cannot always be correctly done. For more information, see the section entitled I/O.

open provides a mode, *O_APPEND*, which causes the file being opened to be positioned at its end. This mode is supported on UNIX Systems 3 and 5, but not UNIX version 7. As with *lseek*, the positioning may not be correct for systems which don't keep track of the last character written to a file.

3.2 Device I/O

3.2.1 Unbuffered I/O to the Console

There are several options available when accessing the console, which are discussed in detail in the Console I/O sections of this chapter and of the system-dependent appendix to this chapter. Here we just want to briefly discuss the line- or character-modes of console I/O as they relate to the unbuffered i/o functions.

Console input can be either line- or character-oriented. With line-oriented input, characters are read from the console into an internal buffer a line at a time, and returned to the program from this buffer. Line buffering of console input is available even when using the so-called "unbuffered" i/o functions.

With character-oriented input, characters are read and returned to the program when they are typed: no buffering of console input occurs.

3.2.2 Unbuffered I/O to Non-Console Devices

Unbuffered I/O to devices other than the console is truly unbuffered.

4. Overview of Console I/O

A program has control over several options relating to console i/o. The primary option allows console input to be either line- or character-oriented, as described below.

On most systems, a program can selectively enable and disable the echoing of typed characters to the screen; this is called the ECHO option. A program can also enable and disable the conversion of carriage return to newline on input and of newline to carriage return-linefeed on output; this is called the CRMOD option.

On some systems, additional options are available. If your system supports additional options, they are discussed in the system dependent appendix to this chapter.

All the console i/o options have default settings, which allow a program to easily access the console without having to set the options itself. In the default mode, console i/o is line-oriented, with ECHO and CRMOD enabled.

A program can easily change the console i/o options, by calling the function *ioctl*.

Console i/o behaves the same on all systems when the console options have their default settings. However, the behavior of console i/o differs from system to system when the options are changed from their default values. Thus, a program requiring machine independence should either use the console in its default mode or be careful how it sets the console options. In the paragraphs below, we will try to point out system dependencies.

4.1 Line-oriented input

With line-oriented input, a program issuing a read request to the console will wait until an entire line has been typed. On some systems a non-UNIX option (NODELAY) is available that will prevent this waiting. If this option is available on your system, it's discussed in the system-dependent appendix to this chapter.

The program need not read an entire line at once; the line will be internally buffered, and characters returned to the program from the buffer, as requested. When the program issues a read request to the console and the buffer is empty, the program will wait until an entire new line has been typed and stored in the internal buffer (again, on some systems programs can disable this wait by setting the non-UNIX NODELAY option).

A single unbuffered read operation can return at most one line.

On most systems, selecting line-oriented console input forces the ECHO option to be enabled. On such systems the program still has control over the CRMOD option. To find out if, on your system,

line-oriented mode always has ECHO enabled, see the system-dependent appendix to this chapter.

4.2 Character-oriented input

The basic idea of character-oriented console input is that a program can read characters from the console without having to wait for an entire line to be entered.

The behavior of character-oriented console input differs from system to system, so programs requiring both machine independence and character-oriented console input have to be careful in their use of the console. However, it is possible to write such programs, although they may not be able to take full advantage of the console i/o features available for a particular system.

There are two varieties of character-oriented console input, named CBREAK and RAW. Their primary difference is that with the console in CBREAK mode, a program still has control over the other console options, whereas with the console in RAW mode it doesn't. In RAW mode, all other console options are reset: ECHO and CRMOD are disabled.

Thus, to some extent RAW mode is simply an abbreviation for 'CBREAK on, all other options off'. However, there are some differences on some systems, as noted below and in this chapter's system-dependent appendix.

The system-dependent appendix to this chapter, which accompanies your manual, presents information about character-oriented console that is specific to your system.

4.2.1 Writing system-independent programs

To write system-independent programs that access the console in character-oriented input mode, the console should be set in RAW mode, and the program should read only a single character at a time from the console. All the non-UNIX options that are supported by some systems should be reset.

The standard i/o functions all read just one character at a time from the console, even when the calling program requests several characters. Thus, programs requiring system independence and character-oriented input can read the console using the standard i/o functions.

Some systems require a program that wants to set console option to first call *ioctl* to fetch the current console options, then modify them as desired, and finally call *ioctl* to reset the new console options. The systems that don't require this don't care if a program first fetches the console options and then modifies them. Thus, a program requiring system-independence and console i/o options other than the default should fetch the current console options before modifying them.

4.3 Using *ioctl*

A program selects console I/O modes using the function *ioctl*. This has the form:

```
#include <sgtty.h>

ioctl(fd, code, arg)
struct sgttyb *arg;
```

The header file *sgtty.h* defines symbolic values for the *code* parameter (which tells *ioctl* what to do) and the structure *sgttyb*.

The parameter *fd* is a file descriptor associated with the console. On UNIX, this parameter defines the file descriptor associated with the device to which the *ioctl* call applies. Here, *ioctl* always applies to the console.

The parameter *code* defines the action to be performed by *ioctl*. It can have these values:

<i>TIOCGETP</i>	Fetch the console parameters and store them in the structure pointed at by <i>arg</i> .
<i>TIOCSETP</i>	Set the console parameters according to the structure pointed at by <i>arg</i> .
<i>TIOCSETN</i>	Equivalent to <i>TIOCSETP</i> .

The argument *arg* points to a structure named *sgttyb* that contains the following fields:

```
int sg_flags;
char sg_erase;
char sg_kill;
```

The order of these fields is system-dependent.

The *sg_flags* field is supported by all systems, while the other fields are not supported by some systems. If these fields are supported on your system, the system-dependent appendix to this chapter that accompanies your manual says so, and describes them.

To set console options, a program should fetch the current state of the *sgtty* fields, using *ioctl*'s *TIOCGETP* option. Then it should modify the fields to the appropriate values and call *ioctl* again, using *ioctl*'s *TIOCSETP* option.

4.4 The *sgtty* fields

4.4.1 The *sg_flags* field

sg_flags contains the following UNIX-compatible flags:

<i>RAW</i>	Set RAW mode (turns off other options). By default, RAW is disabled.
<i>CBREAK</i>	Return each character as soon as typed. By default, CBREAK is disabled.

<i>ECHO</i>	Echo input characters to the display. By default, ECHO is enabled.
<i>CRMOD</i>	Map CR to LF on input; convert LF to CR-LF on output. By default, CRMOD is enabled.

On some systems, other flags are contained in *sg_flags*. If your system supports other flags, they're described in the system-dependent appendix to this chapter that accompanies your manual.

More than one flag can be specified in a single call to *ioctl*; the values are simply 'or'ed together. If the RAW option is selected, none of the other options have any effect.

When the console i/o options are set and RAW and CBREAK are reset, the console is set in line-oriented input mode.

4.5 Examples

4.5.1 Console input using default mode

The following program copies characters from stdin to stdout. The console is in default mode, and assuming these streams haven't been redirected by the operator, the program will read from the keyboard and write to the display. In this mode, the operator can use the operating system's line editing facilities, such as backspace, and characters entered on the keyboard will be echoed to the display. The characters entered won't be returned to the program until the operator depresses carriage return.

```
#include <stdio.h>

main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

4.5.2 Console input - RAW mode

In this example, a program opens the console for standard i/o, sets the console in RAW mode, and goes into a loop, waiting for characters to be read from the console and then processing them. The characters typed by the operator aren't displayed unless the program itself displays them. The input request won't terminate until a character is received. This example assumes that the console is named 'con:'; on systems for which this is not the case, just substitute the appropriate name.

```

#include <stdio.h>
#include <sgtty.h>
main()
{
    int c;
    FILE *fp;
    struct sgttyb stty;
    if ((fp = fopen("con:", "r") == NULL){
        printf("can't open the console\n");
        exit();
    }

    ioctl(fileno(fp), TIOCGETP, &stty);
    stty.sg_flags |= RAW;
    ioctl(fileno(fp), TIOCSETP, &stty);
    for (;;) {
        c = getc(fp);
        ...
    }
}

```

4.5.3 Console input - console in CBREAK + ECHO mode

This example modifies the previous program so that characters read from the console are automatically echoed to the display. The program accesses the console via the standard input device. It uses the function *isatty* to verify that *stdin* is associated with the console; if it isn't, the program reopens *stdin* to the console using the function *freopen*. Again, the console is assumed to be named *con:*.

```

#include <stdio.h>
#include <sgtty.h>
main()
{
    int c;
    struct sgttyb stty;
    if (!isatty(stdin))
        freopen("con:", "r", stdin);
    ioctl(0, TIOCGETP, &stty);
    stty.sg_flags |= CBREAK | ECHO;
    ioctl(0, TIOCSETP, &stty);
    for (;;) {
        c = getchar();
        ...
    }
}

```

5. Overview of Dynamic Buffer Allocation

Several functions are provided for the dynamic allocation and deallocation of buffers from a section of memory called the 'heap'. They are:

<i>malloc</i>	Allocates a buffer
<i>calloc</i>	Allocates a buffer and initializes it to zeroes
<i>realloc</i>	Allocates more space to a previously allocated buffer
<i>free</i>	Releases an allocated buffer for reuse

These standard UNIX functions are described in the System Independent Functions section of this chapter.

In addition, on some systems the UNIX-compatible functions *sbrk* and *brk* are provided that provide a more elementary means to allocate heap space. The *malloc*-type functions call *sbrk* to get heap space, which they then manage.

On some systems, non-UNIX memory allocation functions are also supported. If such functions are supported on your system, they are described in the system-dependent appendix to this chapter that accompanies your manual.

Dynamic allocation of standard i/o buffers

Buffers used for standard i/o are dynamically allocated from the heap unless specific actions are taken by the user's program. Standard i/o calls to dynamically allocate and deallocate buffers can be interspersed with those of the user's program.

Programs which perform standard i/o and which must have absolute control of the heap can explicitly define the buffers to be used by a standard i/o stream.

Where to go from here

For descriptions of the *sbrk* and *brk* functions and, when applicable, non-UNIX memory allocation functions see the System Dependent Functions chapter.

For a discussion of i/o buffer allocation, see the Standard I/O section of the Library Functions Overviews chapter.

For more information on the heap, see the Program Organization section of the Technical Information chapter.

6. Overview of Error Processing

This section discusses error processing which relates to the global integer *errno*. This variable is modified by the standard i/o, unbuffered i/o, and scientific (eg, *sin*, *sqrt*) functions as part of their error processing.

The handling of floating point exceptions (overflow, underflow, and division by zero) is discussed in the Tech Info chapter.

When a standard i/o, unbuffered i/o, or scientific function detects an error, it sets a code in *errno* which describes the error. If no error occurs, the scientific functions don't modify *errno*. If no error occurs, the i/o functions may or may not modify *errno*.

Also, when an error occurs,

- * A standard i/o function returns -1 and sets an error flag for the stream on which the error occurred;
- * An unbuffered i/o function returns -1;
- * A scientific function returns an arbitrary value.

When performing scientific calculations, a program can check *errno* for errors as each function is called. Alternatively, since *errno* is modified only when an error occurs, *errno* can be checked only after a sequence of operations; if it's non-zero, then an error has occurred at some point in the sequence. This latter technique can only be used when no i/o operations occur during the sequence of scientific function calls.

Since *errno* may be modified by an i/o function even if an error didn't occur, a program can't perform a sequence of i/o operations and then check *errno* afterwards to detect an error. Programs performing unbuffered i/o must check the result of each i/o call for an error.

Programs performing standard i/o operations cannot, following a sequence of standard i/o calls, check *errno* to see if an error occurred. However, associated with each open stream is an error flag. This flag is set when an error occurs on the stream and remains set until the stream is closed or the flag is explicitly reset. Thus a program can perform a sequence of standard i/o operations on a stream and then check the stream's error flag. For more details, see the standard i/o overview section.

The following table lists the system-independent values which may be placed in *errno*. These symbolic values are defined in the file *errno.h*. Other, system-dependent, values may also be set in *errno* following an i/o operation; these are error codes returned by the operating system. System dependent error codes are described in the operating system manual for a particular system.

The system-independent error codes and their meanings are:

<i>error code</i>	<i>meaning</i>
ENOENT	File does not exist
E2BIG	Not used
EBADF	Bad file descriptor - file is not open or improper operation requested
ENOMEM	Insufficient memory for requested operation
EEXIST	File already exists on creat request
EINVAL	Invalid argument
ENFILE	Exceeded maximum number of open files
EMFILE	Exceeded maximum number of file descriptors
ENOTTY	Ioctl attempted on non-console
EACCES	Invalid access request
ERANGE	Math function value can't be computed
EDOM	Invalid argument to math function

SYSTEM-INDEPENDENT FUNCTIONS

Chapter Contents

System Independent Functions lib
Index 5
The functions 8

System Independent Functions

This chapter describes in detail the functions which are UNIX-compatible and which are common to all Aztec C packages.

The chapter is divided into sections, each of which describes a group of related functions. Each section has a name, and the sections are ordered alphabetically by name. Following this introduction is a cross reference which lists each function and the name of the section in which it is described.

A section is organized into the following subsections:

TITLE

Lists the name of the section, a phrase which is intended to categorize the functions described in the section, and one or more letters in parentheses which specify the libraries containing the section's functions.

The letters which may appear in parentheses and their corresponding libraries are:

C	c.lib
M	m.lib

On some systems, the actual library name may be a variant on the name given above. For example, on TRSDOS, the libraries are named *c/lib* and *m/lib*.

With *Apprentice C*, the functions are all in the run-time system, and not libraries.

SYNOPSIS

Indicates the types of arguments that the functions described in the section require, and the values they return. For example, the function *atof* converts character strings into double precision numbers. It is listed in the synopsis as

```
double atof(s)
char *s;
```

This means that *atof()* returns a value of type *double* and requires as an argument a pointer to a character string. Since *atof* returns a non-integer value, prior to use of the function it must be declared:

```
double atof();
```

The notation

```
#include "header.h"
```

at the beginning of a synopsis indicates that such a statement should appear at the beginning of any program calling one of the functions described in the section.

On Radio Shack systems, a header file can use either a period or a slash to separate the filename from the extent. That is, the include statement can be as listed above, or

```
#include "header/h"
```

DESCRIPTION

Describes the section's functions.

SEE ALSO

Lists relevant sections. A letter in parentheses may follow a section name. This specifies where the section is located: no letter means that the section is in the current chapter; 'O' means that it's in the Functions Overview chapter; 'S' means that it's in the System Dependent Functions chapter.

DIAGNOSTICS

Describes the error codes that the section's functions may return. The section **ERRORS** in the Functions Overview chapter presents an overview of error processing.

EXAMPLES

Gives examples on use of the section's functions.

Index to System Independent Functions

<i>function</i>	<i>page</i>	<i>description</i>
acos	SIN	compute arccosine
agetc	GETC	get ASCII char from a stream
aputc	PUTC	put ASCII char to a stream
asin	SIN	compute arcsine
atan	SIN	compute arctangent
atan2	SIN	another arctangent function
atof	ATOF	convert char string to a <i>double</i>
atoi	ATOF	convert char string to an <i>int</i>
atol	ATOF	convert char string to a <i>long</i>
calloc	MALLOC	allocate a buffer
ceil	FLOOR	get smallest integer not less than x
clearerr	FERROR	clear error flags on a stream
close	CLOSE	close of unbuffered file/device
cos	SIN	compute cosine
cosh	SINH	compute hyperbolic cosine
cotan	SIN	compute cotangent
creat	CREAT	create a file & open for unbuffered i/o
exp	EXP	compute exponential
fabs	FLOOR	compute absolute value
fclose	FCLOSE	close i/o stream
fdopen	FOPEN	open file descriptor as an i/o stream
feof	FERROR	check for eof on an i/o stream
ferror	FERROR	check for error on an i/o stream
fflush	FCLOSE	flush an i/o stream
fgets	GETS	get a line from an i/o stream
fileno	FERROR	get file descriptor for i/o stream
floor	FLOOR	get largest <i>int</i> not greater than x
fopen	FOPEN	open i/o stream
format	PRINTF	formatting utility for <i>printf</i>
fprintf	PRINTF	format string & send to i/o stream
fputs	PUTS	put char string to i/o stream
fread	FREAD	read binary data from i/o stream
free	MALLOC	release buffer
freopen	FOPEN	reopen i/o stream
frexp	FREXP	get components of a <i>double</i>
fscanf	SCANF	input string from i/o stream & convert
fseek	FSEEK	position i/o stream
ftell	FSEEK	determine position in i/o stream
ftoa	ATOF	convert float/double to char string

fwrite	FREAD	write binary data to i/o stream
getc	GETC	get binary char from i/o stream
getchar	GETC	get ASCII char from stdin
gets	GETS	get ASCII line from stdin
getw	GETW	get ASCII word from stdin
index	STRING	find char in string
ioctl	IOCTL	set mode of device
isalpha, etc.	CTYPE	char classification functions
isatty	IOCTL	is this a console?
ldexp	FREXP	build <i>double</i>
log	EXP	compute natural logarithm
log10	EXP	compute base-10 log
longjmp	SETJMP	non-local goto
lseek	LSEEK	position unbuffered i/o file
malloc	MALLOC	allocate buffer
movmem	MOVMEM	copy a block of memory
modf	FREXP	get components of <i>double</i>
open	OPEN	open file/device for unbuffered i/o
pow	EXP	compute $x^{**}y$
printf	PRINTF	format data and print on stdout
putc	PUTC	put binary char to i/o stream
putchar	PUTC	put ASCII char to stdout
puterr	PUTC	put ASCII char to stderr
puts	PUTS	put ASCII string to stdout
putw	PUTC	put ASCII word to stdout
qsort	QSORT	Quick sort
ran	RAN	compute random number
read	READ	read unbuffered file/device
realloc	MALLOC	reallocate buffer
rename	RENAME	rename file
rindex	STRING	find char in string
scanf	SCANF	input string from stdin & convert
setbuf	SETBUF	set buffer for i/o stream
setjmp	SETJMP	<i>longjmp</i> partner
setmem	MOVMEM	set memory to specified byte
sin	SIN	compute sine
sinh	SINH	compute hyperbolic sine
sprintf	PRINTF	format string into buffer
sqrt	EXP	compute square root
sscanf	SCANF	convert string from buffer
strcat	STRING	concatenate two strings
strcmp	STRING	compare two strings
strcpy	STRING	copy char string
strlen	STRING	get length of char string
strncat	STRING	concatenate strings
strncmp	STRING	compare strings
strncpy	STRING	copy string
swapmem	MOVMEM	swap two blocks of memory

tan	SIN	compute tangent
tanh	SINH	compute hyperbolic tangent
tolower	TOUPPER	convert upper case char to lower
toupper	TOUPPER	convert lower case char to upper
ungetc	UNGETC	return char to i/o stream
unlink	UNLINK	delete file
write	WRITE	unbuffered write of binary data

NAME

atof, *atoi*, *atol* - convert ASCII to numbers
ftoa - convert floating point to ASCII

SYNOPSIS

```
double atof(cp)
char *cp;

atoi(cp)
char *cp;

long atol(cp)
char *cp;

ftoa(val, buf, precision, type)
double val;
char *buf;
int precision, type;
```

DESCRIPTION

atof, *atoi*, and *atol* convert a string of text characters pointed at by the argument *cp* to double, integer, and long representations, respectively.

atof recognizes a string containing leading blanks and tabs, which it skips, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi and *atol* recognize a string containing leading blanks and tabs, which are ignored, then an optional sign, then a string of digits.

ftoa converts a double precision floating point number to ASCII. *val* is the number to be converted and *buf* points to the buffer where the ASCII string will be placed. *precision* specifies the number of digits to the right of the decimal point. *type* specifies the format: 0 for "E" format, 1 for "F" format, 2 for "G" format.

atof and *ftoa* are in the library *m.lib*; the other functions are in *c.lib*.

NAME

close - close a device or file

SYNOPSIS

***close*(*fd*)**

int *fd*;

DESCRIPTION

close closes a device or disk file which is opened for unbuffered i/o.

The parameter *fd* is the file descriptor associated with the file or device. If the device or file was explicitly opened by the program by calling *open* or *creat*, *fd* is the file descriptor returned by *open* or *creat*.

close returns 0 as its value if successful.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *close* fails, it returns -1 and sets an error code in the global integer *errno*.

NAME

`creat` - create a new file

SYNOPSIS

```
creat(name, pmode)
char *name;
int pmode;
```

DESCRIPTION

creat creates a file and opens it for unbuffered, write-only access. If the file already exists, it is truncated so that nothing is in it (this is done by erasing and then creating the file).

creat returns as its value an integer called a "file descriptor". Whenever a call is made to one of the unbuffered i/o functions to access the file, its file descriptor must be included in the function's parameters.

name is a pointer to a character string which is the name of the device or file to be opened. See the I/O overview section for details.

For most systems, *pmode* is optional: if specified, it's ignored. It should be included, however, for programs for which UNIX-compatibility is required, since the UNIX *creat* function requires it. In this case, *pmode* should have the octal value 0666.

For some systems, *pmode* is required and has a special meaning. If it is required for your system, the System Dependent Functions chapter will contain a description of the *creat* function, which will define the meaning.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *creat* fails, it returns -1 as its value and sets a code in the global integer *errno*.

NAME

isalpha, *isupper*, *islower*, *isdigit*, *isalnum*, *isspace*,
ispunct, *isprint*, *isctrl*, *isascii*
 - character classification functions

SYNOPSIS

```
#include "ctype.h"

isalpha(c)
```

...

DESCRIPTION

These macros classify ASCII-coded integer values by table lookup, returning nonzero if the integer is in the category, zero otherwise. *isascii* is defined for all integer values. The others are defined only when *isascii* is true and on the single non-ASCII value EOF (-1).

<i>isalpha</i>	<i>c</i> is a letter
<i>isupper</i>	<i>c</i> is an upper case letter
<i>islower</i>	<i>c</i> is a lower case letter
<i>isdigit</i>	<i>c</i> is a digit
<i>isalnum</i>	<i>c</i> is an alphanumeric character
<i>isspace</i>	<i>c</i> is a space, tab, carriage return, newline, or formfeed
<i>ispunct</i>	<i>c</i> is a punctuation character
<i>isprint</i>	<i>c</i> is a printing character, valued 0x20 (space) through 0x7e (tilde)
<i>isctrl</i>	<i>c</i> is a delete character (0xff) or ordinary control character (value less than 0x20)
<i>isascii</i>	<i>c</i> is an ASCII character, code less than 0x100

NAME

exponential, logarithm, power, square root functions:
exp, *log*, *log10*, *pow*, *sqrt*

SYNOPSIS

```
#include <math.h>
```

```
double exp(x)  
double x;
```

```
double log(x)  
double x;
```

```
double log10(x)  
double x;
```

```
double pow(x, y)  
double x,y;
```

```
double sqrt(x)  
double x;
```

DESCRIPTION

exp returns the exponential function of *x*.

log returns the natural logarithm of *x*; *log10* returns the base 10 logarithm.

pow returns $x ** y$ (*x* to the *y*-th power).

sqrt returns the square root of *x*.

SEE ALSO

Errors (O)

DIAGNOSTICS

If a function can't perform the computation, it sets an error code in the global integer *errno* and returns an arbitrary value; otherwise it returns the computed value without modifying *errno*. The symbolic values which a function can place in *errno* are EDOM, signifying that the argument was invalid, and ERANGE, meaning that the value of the function couldn't be computed. These codes are defined in the file *errno.h*.

The following table lists, for each function, the error codes that can be returned, the function value for that error, and the meaning of the error. The symbolic values are defined in the file *math.h*.

function	error	f(x)	Meaning
exp	ERANGE	HUGE	$x > \text{LOGHUGE}$
"	ERANGE	0.0	$x < \text{LOGTINY}$
log	EDOM	-HUGE	$x \leq 0$
log10	EDOM	-HUGE	$x \leq 0$
pow	EDOM	-HUGE	$x < 0, x=y=0$
"	ERANGE	HUGE	$y \cdot \log(x) > \text{LOGHUGE}$
"	ERANGE	0.0	$y \cdot \log(x) < \text{LOGTINY}$
sqrt	EDOM	0.0	$x < 0.0$

NAME

`fclose`, `fflush` - close or flush a stream

SYNOPSIS

```
#include "stdio.h"
```

```
fclose(stream)
```

```
FILE *stream;
```

```
fflush(stream)
```

```
FILE *stream;
```

DESCRIPTION

fclose informs the system that the user's program has completed its buffered i/o operations on a device or file which it had previously opened (by calling *fopen*). *fclose* releases the control blocks and buffers which it had allocated to the device or file. Also, when a file is being closed, *fclose* writes any internally buffered information to the file.

fclose is called automatically by *exit*.

fflush causes any buffered information for the named output stream to be written to that file. The stream remains open.

If *fclose* or *fflush* is successful, it returns 0 as its value.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

If the operation fails, -1 is returned, and an error code is set in the global integer *errno*.

NAME

feof, *ferror*, *clearerr*, *fileno* - stream status inquiries

SYNOPSIS

```
#include "stdio.h"
```

```
feof(stream)
```

```
FILE *stream;
```

```
ferror(stream)
```

```
FILE *stream;
```

```
clearerr(stream)
```

```
FILE *stream;
```

```
fileno(stream)
```

```
FILE *stream;
```

DESCRIPTION

feof returns non-zero when end-of-file is reached on the specified input stream, and zero otherwise.

ferror returns non-zero when an error has occurred on the specified stream, and zero otherwise. Unless cleared by *clearerr*, the error indication remains set until the stream is closed.

clearerr resets an error indication on the specified stream.

fileno returns the integer file descriptor associated with the stream.

These functions are defined as macros in the file *stdio.h*.

SEE ALSO

Standard I/O (O)

NAME

fabs, *floor*, *ceil* - absolute value, floor, ceiling routines

SYNOPSIS

```
#include <math.h>
```

```
double floor(x)
```

```
double x;
```

```
double ceil(x)
```

```
double x;
```

```
double fabs(x)
```

```
double x;
```

DESCRIPTION

fabs returns the absolute value of *x*.

floor returns the largest integer not greater than *x*.

ceil returns the smallest integer not less than *x*.

NAME

fopen, *freopen*, *fdopen* - open a stream

SYNOPSIS

```
#include "stdio.h"
```

```
FILE *fopen(filename, mode)  
char *filename, *mode;
```

```
FILE *freopen(filename, mode, stream)  
char *filename, *mode;  
FILE *stream;
```

```
FILE *fdopen(fd, mode)  
char *mode;
```

DESCRIPTION

These functions prepare a device or disk file for access by the standard i/o functions; this is called "opening" the device or file. A file or device which has been opened by one of these functions is called a "stream".

If the device or file is successfully opened, these functions return a pointer, called a "file pointer" to a structure of type FILE. This pointer is included in the list of parameters to buffered i/o functions, such as *getc* or *putc*, which the user's program calls to access the stream.

fopen is the most basic of these functions: it simply opens the device or file specified by the *filename* parameter for access specified by the *mode* parameter. These parameters are described below.

freopen substitutes the named device or file for the device or file which was previously associated with the specified stream. It closes the device or file which was originally associated with the stream and returns *stream* as its value. It is typically used to associate devices and files with the preopened streams *stdin*, *stdout*, and *stderr*.

fdopen opens a device or file for buffered i/o which has been previously opened by one of the unbuffered open functions *open* and *creat*. It returns as its value a FILE pointer.

fdopen is passed the file descriptor which was returned when the device or file was opened by *open* or *creat*. It's also passed the *mode* parameter specifying the type of access desired. *mode* must agree with the mode of the open file.

The parameter *filename* is a pointer to a character string which is the name of the device or file to be opened. For details, see the I/O overview section.

mode points to a character string which specifies how the user's program intends to access the stream. The choices are as follows:

<i>mode</i>	<i>meaning</i>
r	Open for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.
w	Open for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created.
a	Open for appending. The calling program is granted write-only access to the stream. The current file position is the character after the last character in the file. If the file does not exist, it is created.
x	Open for writing. The file must not previously exist. This option is not supported by Unix.
r+	Open for reading and writing. Same as "r", but the stream may also be written to.
w+	Open for writing and reading. Same as "w", but the stream may also be read; different from "r+" in the creation of a new file and loss of any previous one.
a+	Open for appending and reading. Same as "a", but the stream may also be read; different from "r+" in file positioning and file creation.
x+	Open for writing and reading. Same as "x" but the file can also be read.

On systems which don't keep track of the last character in a file (for example CP/M and Apple DOS), not all files can be correctly positioned when opened in append mode. See the I/O overview section for details.

SEE ALSO

I/O (O), Standard I/O (O)

DIAGNOSTICS

If the file or device cannot be opened, NULL is returned and an error code is set in the global integer *errno*.

EXAMPLES

The following example demonstrates how *fopen* can be used in a program.


```
#include "stdio.h"

main(argc,argv)
char **argv;
{
    FILE *fopen(), *fp;
    if ((fp = fopen(argv[1], argv[2])) == NULL) {
        printf("You asked me to open %s",argv[1]);
        printf("in the %s mode", argv[2]);
        printf("but I can't!\n");
    } else
        printf("%s is open\n", argv[1]);
}
```

Here is a program which uses *freopen*:

```
#include "stdio.h"
main()
{
    FILE *fp;
    fp = freopen("dskfile", "w+", stdout);
    printf("This message is going to dskfile\n");
}
```

Here is a program which uses *fdopen*:

```
#include "stdio.h"
dopen__it(fd)
int fd; /* value returned by previous call to open */
{
    FILE *fp;
    if ((fp = fdopen(fd, "r+")) == NULL)
        printf("can't open file for r+\n");
    else
        return(fp);
}
```

NAME

fread, fwrite - buffered binary input/output

SYNOPSIS

```
#include "stdio.h"
```

```
int fread(buffer, size, count, stream)
```

```
char *buffer;
```

```
int size, count;
```

```
FILE *stream;
```

```
int fwrite(buffer, size, count, stream)
```

```
char *buffer;
```

```
int size, count;
```

```
FILE *stream;
```

DESCRIPTION

fread performs a buffered input operation and *fwrite* a buffered write operation to the open stream specified by the parameter *stream*.

buffer is the address of the user's buffer which will be used for the operation.

The function reads or writes *count* items, each containing *size* bytes, from or to the stream.

fread and *fwrite* perform i/o using the functions *getc* and *putc*; thus, no translations occur on the data being transferred.

The function returns as its value the number of items actually read or written.

SEE ALSO

Standard I/O (O), Errors (O), fopen, ferror

DIAGNOSTICS

fread and *fwrite* return 0 upon end of file or error. The functions *feof* and *ferror* can be used to distinguish between the two. In case of an error, the global integer *errno* contains a code defining the error.

EXAMPLE

This is the code for reading ten integers from file 1 and writing them again to file 2. It includes a simple check that there are enough two-byte items in the first file:

```
#include "stdio.h"

main()
{
    FILE *fp1, *fp2, *fopen();
    char *buf;
    int size = 2, count = 10;

    fp1 = fopen("file1","r");
    fp2 = fopen("file2","w");
    if (fread(buf, size, count, fp1) != count)
        printf("Not enough integers in file1\n");
    fwrite(buf, size, count, fp2);
}
```

NAME

frexp, *ldexp*, *modf* - build and unbuild real numbers

SYNOPSIS

```
#include <math.h>
```

```
double frexp(value, eptr)
```

```
double value;
```

```
int *eptr;
```

```
double ldexp(value, exp)
```

```
double value;
```

```
double modf(value, iptr)
```

```
double value, *iptr;
```

DESCRIPTION

Given *value*, *frexp* computes integers *x* and *n* such that $value = x * 2^{**n}$. *x* is returned as the value of *frexp*, and *n* is stored in the *int* field pointed at by *eptr*.

ldexp returns the double quantity $value * 2^{**exp}$.

modf returns as its value the positive fractional part of *value* and stores the integer part in the double field pointed at by *iptr*.

NAME

fseek, *ftell* - reposition a stream

SYNOPSIS

```
#include "stdio.h"

int fseek(stream, offset, origin)
FILE *stream;
long offset;
int origin;

long ftell(stream)
FILE *stream;
```

DESCRIPTION

fseek sets the "current position" of a file which has been opened for buffered i/o. The current position is the byte location at which the next input or output operation will begin.

stream is the stream identifier associated with the file, and was returned by *fopen* when the file was opened.

offset and *origin* together specify the current position: the new position is at the signed distance *offset* bytes from the beginning, current position, or end of the file, depending on whether *origin* is 0, 1, or 2, respectively.

offset can be positive or negative, to position after or before the specified origin, respectively, with the limitation that you can't seek before the beginning of the file.

For some operating systems (for example, CP/M and Apple DOS) a file may not be able to be correctly positioned relative to its end. See the overview sections I/O and STANDARD I/O for details.

If *fseek* is successful, it will return zero.

ftell returns the number of bytes from the beginning to the current position of the file associated with *stream*.

SEE ALSO

Standard I/O (O), I/O (O), *lseek*

DIAGNOSTICS

fseek will return -1 for improper seeks. In this case, an error code is set in the global integer *errno*.

EXAMPLE

The following routine is equivalent to opening a file in "a+" mode:

```
a__plus(filename)
char *filename;
{
    FILE *fp, *fopen();
    if ((fp = fopen(filename, r+)) == NULL)
        fp = fopen(filename, w+);
    fseek(fp, 0L, 2); /* position 1 byte past
                      last character */
}
```

To set the current position back 5 characters before the present current position, the following call can be used:

```
fseek(fp, -5L, 1)
```

NAME

getc, agetc, getchar, getw

SYNOPSIS

```
#include "stdio.h"

int getc(stream)
FILE *stream;

int agetc(stream)    /* non-Unix function */
FILE *stream;

int getchar()

int getw(stream)
FILE *stream;
```

DESCRIPTION

getc returns the next character from the specified input stream.

agetc is used to access files of text. It generally behaves like *getc* and returns the next character from the named input stream. It differs from *getc* in the following ways:

- * It translates end-of-line sequences (eg, carriage return on Apple DOS; carriage return-line feed on CP/M) to a single newline ('\n') character.
- * It translates an end-of-file sequence (eg, a null character on Apple DOS; a control-z character on CP/M) to EOF;
- * It ignores null characters (' ') on all systems except Apple DOS;
- * On some systems, the most significant bit of each character returned is set to zero.

agetc is not a UNIX function. It is, however, provided with all Aztec C packages, and provides a convenient, system-independent way for programs to read text.

getchar returns text characters from the standard input stream (stdin). It is implemented as the call *agetc(stdin)*.

getw returns the next word from the specified input stream. It returns EOF (-1) upon end-of-file or error, but since that is a good integer value, *feof* and *ferror* should be used to check the success of *getw*. It assumes no special alignment in the file.

SEE ALSO

I/O (O), Standard I/O (O), fopen, fclose

DIAGNOSTICS

These functions return EOF (-1) at end of file or if an error occurs. The functions *feof* and *ferror* can be used to distinguish the two. In the latter case, an error code is set in the global

integer *errno*.

NAME

gets, *fgets* - get a string from a stream

SYNOPSIS

```
#include "stdio.h"
```

```
char *gets(s)
```

```
char *s;
```

```
char *fgets(s, n, stream)
```

```
char *s;
```

```
FILE *stream;
```

DESCRIPTION

gets reads a string of characters from the standard input stream, *stdin*, into the buffer pointed by *s*. The input operation terminates when either a newline character (`\\n`) or end of file is encountered.

fgets reads characters from the specified input stream into the buffer pointer at by *s* until either (1) *n*-1 characters have been read, (2) a newline character (`\\n`) is read, or (3) end of file or an error is detected on the stream.

Both functions return *s*, except as noted below.

gets and *fgets* differ in their handling of the newline character: *gets* doesn't put it in the caller's buffer, while *fgets* does. This is the behavior of these functions under UNIX.

These functions get characters using *agetc*; thus they can only be used to get characters from devices and files which contain text characters.

SEE ALSO

I/O (O), Standard I/O (O), *ferror*

DIAGNOSTICS

gets and *fgets* return the pointer `NULL` (0) upon reaching end of file or detecting an error. The functions *feof* and *ferror* can be used to distinguish the two. In the latter case, an error code is placed in the global integer *errno*.

NAME

ioctl, *isatty* - device i/o utilities

SYNOPSIS

```
#include "sgtty.h"
```

```
ioctl(fd, cmd, stty)
```

```
struct sgttyb *stty;
```

```
isatty(fd)
```

DESCRIPTION

ioctl sets and determines the mode of the console.

For more details on *ioctl*, see the overview section on console I/O.

isatty returns non-zero if the file descriptor *fd* is associated with the console, and zero otherwise.

SEE ALSO

Console I/O (O)

NAME

lseek - change current position within file

SYNOPSIS

```
long int lseek(fd, offset, origin)
int fd, origin;
long offset;
```

DESCRIPTION

lseek sets the current position of a file which has been opened for unbuffered i/o. This position determines where the next character will be read or written.

fd is the file descriptor associated with the file.

The current position is set to the location specified by the *offset* and *origin* parameters, as follows:

- * If *origin* is 0, the current position is set to *offset* bytes from the beginning of the file.
- * If *origin* is 1, the current position is set to the current position plus *offset*.
- * If *origin* is 2, the current position is set to the end of the file plus *offset*.

The *offset* can be positive or negative, to position after or before the specified *origin*, respectively.

Positioning of a file relative to its end (that is, calling *lseek* with *origin* set to 2) cannot always be correctly done on all systems (for example, CP/M and Apple DOS). See the section entitled I/O for details.

If *lseek* is successful, it will return the new position in the file (in bytes from the beginning of the file).

SEE ALSO

Unbuffered I/O (O), I/O (O)

DIAGNOSTICS

If *lseek* fails, it will return -1 as its value and set an error code in the global integer *errno*. *errno* is set to EBADF if the file descriptor is invalid. It will be set to EINVAL if the *offset* parameter is invalid or if the requested position is before the beginning of the file.

EXAMPLES

1. To seek to the beginning of a file:

```
lseek(fd, 0L, 0);
```

lseek will return the value zero (0) since the current position in the file is character (or byte) number zero.

2. To seek to the character following the last character in the file:

```
pos = lseek(fd, 0L, 2);
```

The variable *pos* will contain the current position of the end of file, plus one.

3. To seek backward five bytes:

```
lseek(fd, -5L, 1);
```

The third parameter, 1, sets the origin at the current position in the file. The offset is -5. The new position will be the origin plus the offset. So the effect of this call is to move backward a total of five characters.

4. To skip five characters when reading in a file:

```
read(fd, buf, count);  
lseek(fd, 5L, 1);  
read(fd, buf, count);
```

NAME

malloc, calloc, realloc, free - memory allocation

SYNOPSIS

```
char *malloc(size)
unsigned size;

char *calloc(nelem, elemsize)
unsigned nelem, elemsize;

char *realloc(ptr, size)
char *ptr;
unsigned size;

free(ptr)
char *ptr;
```

DESCRIPTION

These functions are used to allocate memory from the "heap", that is, the section of memory available for dynamic storage allocation.

malloc allocates a block of *size* bytes, and returns a pointer to it.

calloc allocates a single block of memory which can contain *nelem* elements, each *elemsize* bytes big, and returns a pointer to the beginning of the block. Thus, the allocated block will contain (*nelem* * *elemsize*) bytes. The block is initialized to zeroes.

realloc changes the size of the block pointed at by *ptr* to *size* bytes, returning a pointer to the block. If necessary, a new block will be allocated of the requested size, and the data from the original block moved into it. The block passed to *realloc* can have been freed, provided that no intervening calls to *calloc*, *malloc*, or *realloc* have been made.

free deallocates a block of memory which was previously allocated by *malloc*, *calloc*, or *realloc*; this space is then available for reallocation. The argument *ptr* to *free* is a pointer to the block.

malloc and *free* maintain a circular list of free blocks. When called, *malloc* searches this list beginning with the last block freed or allocated coalescing adjacent free blocks as it searches. It allocates a buffer from the first large enough free block that it encounters. If this search fails, it calls *sbrk* to get more memory for use by these functions.

SEE ALSO

Memory Usage (O), break (S)

DIAGNOSTICS

malloc, *calloc* and *realloc* return a null pointer (0) if there is no available block of memory.

free returns -1 if it's passed an invalid pointer.

NAME

movmem, setmem, swapmem

SYNOPSIS

```
movmem(src, dest, length)      /* non-Unix function */
char *src, *dest;
int length;

setmem(area,length,value)      /* non-Unix function */
char *area;

swapmem(s1, s2, len)           /* non-Unix function */
char *s1, *s2;
```

DESCRIPTION

movmem copies *length* characters from the block of memory pointed at by *src* to that pointed at by *dest*.

movmem copies in such a way that the resulting block of characters at *dest* equals the original block at *src*.

setmem sets the character *value* in each byte of the block of memory which begins at *area* and continues for *length* bytes.

swapmem swaps the blocks of memory pointed at by *s1* and *s2*. The blocks are *len* bytes long.

NAME

open

SYNOPSIS

#include "fcntl.h"

```
open(name, mode)    /* calling sequence on most systems */
char *name;
```

```
/* calling sequence on some systems (see below): */
open(name, mode, param3)
char *name;
```

DESCRIPTION

open opens a device or file for unbuffered i/o. It returns an integer value called a file descriptor which is used to identify the file or device in subsequent calls to unbuffered i/o functions.

name is a pointer to a character string which is the name of the device or file to be opened. For details, see the overview section I/O.

mode specifies how the user's program intends to access the file. The choices are as follows:

<i>mode</i>	<i>meaning</i>
O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read and write
O_CREAT	Create file, then open it
O_TRUNC	Truncate file, then open it
O_EXCL	Cause open to fail if file already exists; used with O_CREAT
O_APPEND	Position file for appending data

These open modes are integer constants defined in the files *fcntl.h*. Although the true values of these constants can be used in a given call to open, use of the symbolic names ensures compatibility with UNIX and other systems.

The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, and O_RDWR in the mode parameter. The three remaining values are optional. They may be included by adding them to the mode parameter, as in the examples below.

By default, the open will fail if the file to be opened does not exist. To cause the file to be created when it does not already exist, specify the O_CREAT option. If O_EXCL is given in addition to O_CREAT, the open will fail if the file already exists; otherwise, the file is created.

If the `O_TRUNC` option is specified, the file will be truncated so that nothing is in it. The truncation is performed by simply erasing the file, if it exists, and then creating it. So it is not an error to use this option when the file does not exist.

Note that when `O_TRUNC` is used, `O_CREAT` is not needed.

If `O_APPEND` is specified, the current position for the file (that is, the position at which the next data transfer will begin) is set to the end of the file. For systems which don't keep track of the last character written to a file (for example, CP/M and Apple DOS), this positioning cannot always be correctly done. See the I/O overview section for details. Also, this option is not supported by UNIX.

param3 is not needed or used on many systems. If it is needed for your system, the System Dependent Library Functions chapter will contain a description of the *open* function, which will define this parameter.

If *open* does not detect an error, it returns an integer called a "file descriptor." This value is used to identify the open file during unbuffered i/o operations. The file descriptor is very different from the file pointer which is returned by *fopen* for use with buffered i/o functions.

SEE ALSO

I/O (O), Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *open* encounters an error, it returns -1 and sets the global integer *errno* to a symbolic value which identifies the error.

EXAMPLES

1. To open the file, *testfile*, for read-only access:

```
fd = open("testfile", O_RDONLY);
```

If *testfile* does not exist *open* will just return -1 and set *errno* to *ENOENT*.

2. To open the file, *sub1*, for read-write access:

```
fd = open("sub1", O_RDWR+O_CREAT);
```

If the file does not exist, it will be created and then opened.

3. The following program opens a file whose name is given on the command line. The file must not already exist.

```
main(argc, argv)
char **argv;
{
    int fd;

    fd = open(*++argv, O__WRONLY+O__CREAT+O__EXCI
    if (fd = -1) {
        if (errno == EEXIST)
            printf("file already exists\n");
        else if (errno == ENOENT)
            printf("unable to open file\n");
        else
            printf("open error\n");
    }
}
```

NAME

printf, fprintf, sprintf, format
- formatted output conversion functions

SYNOPSIS

```
#include "stdio.h"

printf(fmt [,arg] ...)
char *fmt;

fprintf(stream, fmt [,arg] ...)
FILE *stream;
char *fmt;

sprintf(buffer, fmt [,arg] ...)
char *buffer, *fmt;

format(func, fmt, argptr)
int (*func)();
char *fmt;
unsigned *argptr;
```

DESCRIPTION

These functions convert and format their arguments (*arg* or *argptr*) according to the format specification *fmt*. They differ in what they do with the formatted result:

printf outputs the result to the standard output stream, *stdout*;

fprintf outputs the result to the stream specified in its first argument, *stream*;

sprintf places the result in the buffer pointed at by its first argument, *buffer*, and terminates the result with the null character, ' '.

format calls the function *func* with each character of the result. In fact, *printf*, *fprintf*, and *sprintf* call *format* with each character that they generate.

These functions are in both *c.lib* and *m.lib*, the difference being that the *c.lib* versions don't support floating point conversions. Hence, if floating point conversion is required, the *m.lib* versions must be used. If floating point conversion isn't required, either version can be used. To use *m.lib*'s version, *m.lib* must be specified before *c.lib* at the time the program is linked.

The character string pointed at by the *fmt* parameter, which directs the print functions, contains two types of items: ordinary characters, which are simply output, and conversion specifications, each of which causes the conversion and output of the next successive *arg*.

A conversion specification begins with the character % and continues with:

- * An optional minus sign (-) which specifies left adjustment of the converted value in the output field;
- * An optional digit string specifying the 'field width' for the conversion. If the converted value has fewer characters than this, enough blank characters will be output to make the total number of characters output equals the field width. If the converted value has more characters than the field width, it will be truncated. The blanks are output before or after the value, depending on the presence or absence of the left- adjustment indicator. If the field width digits have a leading 0, 0 is used as a pad character rather than blank.
- * An optional period, '.', which separates the field width from the following field;
- * An optional digit string specifying a precision; for floating point conversions, this specifies the number of digits to appear after the decimal point; for character string conversions, this specifies the maximum number of characters to be printed from a string;
- * Optionally, the character l, which specifies that a conversion which normally is performed on an *int* is to be performed on a *long*. This applies to the d, o, and x conversions.
- * A character which specifies the type of conversion to be performed.

A field width or precision may be * instead of a number, specifying that the next available *arg*, which must be an *int*, supplies the field width or precision.

The conversion characters are:

- | | |
|------------|---|
| d, o, or x | The <i>int</i> in the corresponding <i>arg</i> is converted to decimal, octal, or hexadecimal notation, respectively, and output; |
| u | The unsigned integer <i>arg</i> is converted to decimal notation; |
| c | The character <i>arg</i> is output. Null characters are ignored; |
| s | The characters in the string pointed at by <i>arg</i> are output until a null character or the number of characters indicated by the precision is reached. If the precision is zero or missing, all characters in the string, up to the terminating null, are output; |
| f | The float or double <i>arg</i> is converted to decimal notation in the style '[-]ddd.ddd'. The number |

of d's after the decimal point is equal to the precision given in the conversion specification. If the precision is missing, it defaults to six digits. If the precision is explicitly 0, the decimal point is also not printed.

e The float or double *arg* is converted to the style '[-]d.ddde[-]dd', where there is one digit before the decimal point and the number after is equal to the precision given. If the precision is missing, it defaults to six digits.

g The float or double *arg* is printed in style d, f, or e, whichever gives full precision in minimum space.

% Output a %. No argument is converted.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. The following program fragment:

```
char *name; float amt;
printf("your total, %s, is $%f\n", name, amt);
```

will print a message of the form

```
your total, Alfred, is $3.120000
```

Since the precision of the %f conversion wasn't specified, it defaulted to six digits to the right of the decimal point.

2. This example modifies example 1 so that the field width for the %s conversion is three characters, and the field width and precision of the %f conversion are 10 and 2, respectively. The %f conversion will also use 0 as a pad character, rather than blank.

```
char *name; float amt;
printf("your total, %3s, is $%10.2f\n", name, amt);
```

3. This example modifies example 2 so that the field width of the %s conversion and the precision of the %f conversion are taken from the variables *nw* and *ap*:

```
char *name; float amt; int nw, ap;
printf("your total %*s,is $%10.*f\n",nw,name,ap,amt);
```

4. This example demonstrates how to use the *format* function by listing *printf*, which calls *format* with each character that it generates.

```
printf(fmt,args)
char *fmt; unsigned args;
{
    extern int putchar();
    format(putchar,fmt,&args);
}
```

NAME

`putc`, `aputc`, `putchar`, `putw`, `puterr`
 - put character or word to a stream

SYNOPSIS

```
#include "stdio.h"

putc(c, stream)
char c;
FILE *stream;

aputc(c, stream)          /* non-Unix function */
char c;
FILE *stream;

putchar(c)
char c;

putw(w, stream)
FILE *stream;

puterr(c)                 /* non-Unix function */
char c;
```

DESCRIPTION

putc writes the character *c* to the named output stream. It returns *c* as its value.

aputc is used to write text characters to files and devices. It generally behaves like *putc*, and writes a single character to a stream. It differs from *putc* as follows:

- * When a newline character is passed to *aputc*, an end-of-line sequence (eg, carriage return followed by line feed on CP/M, and carriage return only on Apple DOS) is written to the stream;
- * The most significant bit of a character is set to zero before being written to the stream.
- * *aputc* is not a UNIX function. It is, however, supported on all Aztec C systems, and provides a convenient, system-independent way for a program to write text.
- * *putchar* writes the character *c* to the standard output stream, `stdout`. It's identical to *aputc*(*c*, `stdout`).
- * *putw* writes the word *w* to the specified stream. It returns *w* as its value. *putw* neither requires nor causes special alignment in the file.
- * *puterr* writes the character *c* to the standard error stream, `stderr`. It's identical to *aputc*(*c*, `stderr`). It is not a UNIX function.

SEE ALSO

Standard I/O

DIAGNOSTICS

These functions return EOF (-1) upon error. In this case, an error code is set in the global integer *errno*.

NAME

`puts`, `fputs` - put a character string on a stream

SYNOPSIS

```
#include "stdio.h"
```

```
puts(s)
```

```
char *s;
```

```
fputs(s, stream)
```

```
char *s;
```

```
FILE *stream;
```

DESCRIPTION

puts writes the null-terminated string *s* to the standard output stream, `stdout`, and then an end-of-line sequence. It returns a non-negative value if no errors occur.

fputs copies the null-terminated string *s* to the specified output stream. It returns 0 if no errors occur.

Both functions write to the stream using *aputc*. Thus, they can only be used to write text. See the `PUTC` section for more details on *aputc*.

Note that *puts* and *fputs* differ in this way: On encountering a newline character, *puts* writes an end-of-line sequence and *fputs* doesn't.

SEE ALSO

Standard I/O (`O`), `putc`

DIAGNOSTICS

If an error occurs, these functions return EOF (-1) and set an error code in the global integer *errno*.

NAME

qsort - sort an array of records in memory

SYNOPSIS

```
qsort(array, number, width, func)
char *array;
unsigned number;
unsigned width;
int (*func)();
```

DESCRIPTION

qsort sorts an array of elements using Hoare's Quicksort algorithm. *array* is a pointer to the array to be sorted; *number* is the number of record to be sorted; *width* is the size in bytes of each array element; *func* is a pointer to a function which is called for a comparison of two array elements.

func is passed pointers to the two elements being compared. It must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The Aztec linker, LN, can generate a file of text containing a symbol table for a program. Each line of the file contains an address at which a symbol is located, followed by a space, followed by the symbol name. The following program reads such a symbol table from the standard input, sorts it by address, and writes it to standard output.

```
#include "stdio.h"
#define MAXLINES 2000
#define LINESIZE 16
char *lines[MAXLINES], *malloc();

main()
{
    int i,numlines, cmp();
    char buf[LINESIZE];

    for (numlines=0; numlines<MAXLINES; ++numlines){
        if (gets(buf) == NULL)
            break;
        lines[numlines] = malloc(LINESIZE);
        strcpy(lines[numlines], buf);
    }
    qsort(lines, numlines, 2, cmp);
    for (i = 0; i < numlines; ++i)
        printf("%s\n", lines[i]);
}

cmp(a,b)
char **a, **b;
{
    return strcmp(*a, *b);
}
```

NAME

ran - random number generator

SYNOPSIS

double ran()

DESCRIPTION

ran returns as its value a random number between 0.0 and 1.0.

NAME

read - read from device or file without buffering

SYNOPSIS

```
read (fd, buf, bufsize)
int fd, bufsize; char *buf;
```

DESCRIPTION

read reads characters from a device or disk file which has been previously opened by a call to *open* or *creat*. In most cases, the information is read directly into the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer into which the information is to be placed.

bufsize is the number of characters to be transferred.

If *read* is successful, it returns as its value the number of characters transferred.

If the returned value is zero, then end-of-file has been reached, immediately, with no bytes read.

SEE ALSO

Unbuffered I/O (O), open, close

DIAGNOSTICS

If the operation isn't successful, *read* returns -1 and places a code in the global integer *errno*.

NAME

rename - rename a disk file

SYNOPSIS

```
rename(oldname, newname)      /* non-Unix function */  
char *oldname,*newname;
```

DESCRIPTION

rename changes the name of a file.

oldname is a pointer to a character array containing the old file name, and *newname* is a pointer to a character array containing the new name of the file.

If successful, *rename* returns 0 as its value; if unsuccessful, it returns -1.

If a file with the new name already exists, *rename* sets E_EXIST in the global integer *errno* and returns -1 as its value without renaming the file.

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include "stdio.h"
```

```
scanf(format [,pointer] ...)
```

```
char *format;
```

```
fscanf(stream, format [,pointer] ...)
```

```
FILE *stream;
```

```
char *format;
```

```
sscanf(buffer, format [,pointer] ...)
```

```
char *buffer, *format;
```

DESCRIPTION

These functions convert a string or stream of text characters, as directed by the control string pointed at by the *format* parameter, and place the results in the fields pointed at by the *pointer* parameters.

The functions get the text from different places:

scanf gets text from the standard input stream, *stdin*;

fscanf gets text from the stream specified in its first parameter, *stream*;

sscanf gets text from the buffer pointed at by its first parameter, *buffer*.

The scan functions are in both *c.lib* and *m.lib*, the difference being that the *c.lib* versions don't support floating point conversions. Hence, if floating point conversion is required, the *m.lib* versions must be used. If floating point conversions aren't required, either version can be used. To use *m.lib*'s version, *m.lib* must be specified before *c.lib* when the program is linked.

The control string pointed at by *format* contains the following 'control items':

- * Conversion specifications;
- * 'White space' characters (space, tab newline);
- * Ordinary characters; that is, characters which aren't part of a conversion specification and which aren't white space.

A scan function works its way through a control string, trying to match each control item to a portion of the input stream or buffer. During the matching process, it fetches characters one at a time from the input. When a character is fetched which isn't appropriate for the control item being matched, the scan function pushes it back into the input stream or buffer and

finishes processing the current control item. This pushing back frequently gives unexpected results when a stream is being accessed by other i/o functions, such as *getc*, as well as the scan function. The examples below demonstrate some of the problems that can occur.

The scan function terminates when it first fails to match a control item or when the end of the input stream or buffer is reached. It returns as its value the number of matched conversion specifications, or EOF if the end of the input stream or buffer was reached.

Matching 'white space' characters

When a white space character is encountered in the control string, the scan function fetches input characters until the first non-white-space character is read. The non-white-space character is pushed back into the input and the scan function proceeds to the next item in the control string.

Matching ordinary characters

If an ordinary character is encountered in the control string, the scan function fetches the next input character. If it matches the ordinary character, the scan function simply proceeds to the next control string item. If it doesn't match, the scan function terminates.

Matching conversion specifications

When a conversion specification is encountered in the control string, the scan function first skips leading white space on the input stream or buffer. It then fetches characters from the stream or buffer until encountering one that is inappropriate for the conversion specification. This character is pushed back into the input.

If the conversion specification didn't request assignment suppression (discussed below), the character string which was read is converted to the format specified by the conversion specification, the result is placed in the location pointed at by the current pointer argument, and the next pointer argument becomes current. The scan function then proceeds to the next control string item.

If assignment suppression was requested by the conversion specification, the scan function simply ignores the fetched input characters and proceeds to the next control item.

Details of input conversion

A conversion specification consists of:

- * The character '%', which tells the scan function that it

has encountered a conversion specification;

- * Optionally, the assignment suppression character '*';
- * Optionally, a 'field width'; that is, a number specifying the maximum number of characters to be fetched for the conversion;
- * A conversion character, specifying the type of conversion to be performed.

If the assignment suppression character is present in a conversion specification, the scan function will fetch characters as if it was going to perform the conversion, ignore them, and proceed to the next control string item.

The following conversion characters are supported:

- % A single '%' is expected in the input; no assignment is done.
- d A decimal integer is expected; the input digit string is converted to binary and the result placed in the *int* field pointed at by the current *pointer* argument;
- o An octal integer is expected; the corresponding *pointer* should point to an *int* field in which the converted result will be placed;
- x A hexadecimal integer is expected; the converted value will be placed in the *int* field pointed at by the current *pointer* argument;
- s A sequence of characters delimited by white space characters is expected; they, plus a terminating null character, are placed in the character array pointed at by the current *pointer* argument.
- c A character is expected. It is placed in the *char* field pointed at by the current *pointer*. The normal skip over leading white space is not done; to read a single char after skipping leading white space, use '%ls'. The field width parameter is ignored, so this conversion can be used only to read a single character.
- [A sequence of characters, optionally preceded by white space but not terminated by white space is expected. The input characters, plus a terminating null character, are placed in the character array pointed at by the current *pointer* argument. The left bracket is followed by:
 - * Optionally, a '^' or '~' character;
 - * A set of characters;
 - * A right bracket, ']'.

If the first character in the set isn't ^ or ~, the set specifies characters which are allowed; characters are fetched from the input until one is read which isn't in the set.

If the first character in the set is ^ or ~, the set specifies characters which aren't allowed; characters are fetched from the input until one is read which is in the set.

- e* A floating point number is expected. The input string is converted to floating point format and stored in the *float* field pointed at by the current *pointer* argument. The input format for floating point numbers consists of an optionally signed string of digits, possibly containing a decimal point, optionally followed by an exponent field consisting of an E or e followed by an optionally signed digit.

The conversion characters d, o, and x can be capitalized or preceded by *l* to indicate that the corresponding *pointer* is to a *long* rather than an *int*. Similarly, the conversion characters e and f can be capitalized or preceded by *l* to indicate that the corresponding *pointer* is to a *double* rather than a *float*.

The conversion characters o, x, and d can be optionally preceded by *h* to indicate that the corresponding *pointer* is to a *short* rather than an *int*. Since *short* and *int* fields are the same in Aztec C, this option has no effect.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. In this program fragment, *scanf* is used to read values for the int x, the float y, and a character string into the char array z:

```
int x; float y; char z[50];
scanf("%d%f%s", &x, &y, z);
```

The input line

```
32 75.36e-1 rufus
```

will assign 32 to x, 7.536 to y, and "rufus " to z. *scanf* will return 3 as its value, signifying that three conversion specifications were matched.

The three input strings must be delimited by 'white space' characters; that is, by blank, tab, and newline characters. Thus, the three values could also be entered on separate

lines, with the white space character newline used to separate the values.

2. This example discusses the problems which may arise when mixing *scanf* and other input operations on the same stream.

In the previous example, the character string entered for the third variable, *z*, must also be delimited by white space characters. In particular, it must be terminated by a space, tab, or newline character. The first such character read by *scanf* while getting characters for *z* will be 'pushed back' into the standard input stream. When another read of *stdin* is made later, the first character returned will be the white space character which was pushed back.

This 'pushing back' can lead to unexpected results for programs that read *stdin* with functions in addition to *scanf*. Suppose that the program in the first example wants to issue a *gets* call to read a line from *stdin*, following the *scanf* to *stdin*. *scanf* will have left on the input stream the white space character which terminated the third value read by *scanf*. If this character is a newline, then *gets* will return a null string, because the first character it reads is the pushed back newline, the character which terminates *gets*. This is most likely not what the program had in mind when it called *gets*.

It is usually unadvisable to mix *scanf* and other input operations on a single stream.

3. This example discusses the behavior of *scanf* when there are white space characters in the control string.

The control string in the first example was "%d%f%s". It doesn't contain or need any white space, since *scanf*, when attempting to match a conversion specification, will skip leading white space. There's no harm in having white space before the %d, between the %d and %f, or between the %f and %s. However, placing a white space character after the %s can have unexpected results. In this case, *scanf* will, after having read a character string for *z*, keep reading characters until a non-white-space character is read. This forces the operator to enter, after the three values for *x*, *y*, and *z*, a non-white space character; until this is done, *scanf* will not terminate.

The programmer might place a newline character at the end of a control string, mistakenly thinking that this will circumvent the problem discussed in example 2. One might think that *scanf* will treat the newline as it would an

ordinary character in the control string; that is, that *scanf* will search for, and remove, the terminating newline character from the input stream after it has matched the *z* variable. However, this is incorrect, and should be remembered as a common misinterpretation.

4. *scanf* only reads input it can match. If, for the first example, the input line had been

```
32 rufus 75.36e-1
```

scanf would have returned with value 1, signifying that only one conversion specification had been matched. *x* would have the value 32, *y* and *z* would be unchanged. All characters in the input stream following the 32 would still be in the input stream, waiting to be read.

5. One common problem in using *scanf* involves mismatching conversion specifications and their corresponding arguments. If the first example had declared *y* to be a double, then one of the following statements would have been required:

```
scanf("%d%lf%s", &x, &y, z);
```

or

```
scanf("%d%F%s", &x, &y, z);
```

to tell *scanf* that the floating point variable was a double rather than a float.

6. Another common problem in using *scanf* involves passing *scanf* the value of a variable rather than its address. The following call to *scanf* is incorrect:

```
int x; float y; char z[50];
scanf("%d%f%s", x, y, z);
```

scanf has been passed the value contained in *x* and *y*, and the address of *z*, but it requires the address of all three variables. The "address of" operator, *&*, is required as a prefix to *x* and *y*. Since *z* is an array, its address is automatically passed to *scanf*, so *z* doesn't need the *&* prefix, although it won't hurt if it is given.

7. Consider the following program fragment:

```
int x; float y; char z[50];
scanf("%2d%f%*d%[1234567890]", &x, &y, z);
```

When given the following input:

```
12345 678 90a65
```

scanf will assign 12 to *x*, 345.0 to *y*, skip '678', and place

the string '90 ' in z. The next call to *getchar* will return 'a'.

NAME

`setbuf` - assign buffer to a stream

SYNOPSIS

```
#include "stdio.h"
```

```
setbuf(stream, buf)
```

```
FILE *stream;
```

```
char *buf;
```

DESCRIPTION

setbuf defines the buffer that's to be used for the i/o stream *stream*. If *buf* is not a NULL pointer, the buffer that it points at will be used for the stream instead of an automatically allocated buffer. If *buf* is a NULL pointer, the stream will be completely unbuffered.

When *buf* is not NULL, the buffer it points at must contain BUFSIZ bytes, where BUFSIZ is defined in *stdio.h*.

setbuf must be called after the stream has been opened, but before any read or write operations to it are made.

If the user's program doesn't specify the buffer to be used for a stream, the standard i/o functions will dynamically allocate a buffer for the stream, by calling the function *malloc*, when the first read or write operation is made on the stream. Then, when the stream is closed, the dynamically allocated buffer is freed by calling *free*.

SEE ALSO

Standard I/O (O), *malloc*

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include "setjmp.h"
```

```
setjmp(env)
```

```
jmp_buf env;
```

```
longjmp(env, val)
```

```
jmp_buf env;
```

DESCRIPTION

These functions are useful for dealing with errors encountered by the low-level functions of a program.

setjmp saves its stack environment in the memory block pointed at by *env* and returns 0 as its value.

longjmp causes execution to continue as if the last call to *setjmp* was just terminating with value *val*. *val* cannot be zero.

The parameter *env* is a pointer to a block of memory which can be used by *setjmp* and *longjmp*. The block must be defined using the typedef *jmp_buf*.

WARNING

longjmp must not be called without *env* having been initialized by a call to *setjmp*. It also must not be called if the function that called *setjmp* has since returned.

EXAMPLE

In the following example, the function *getall* builds a record pertaining to a customer and returns the pointer to the record if no errors were encountered and 0 otherwise.

getall calls other functions which actually build the record. These functions in turn call other functions, which in turn ...

getall defines, by calling *setjmp*, a point to which these functions can branch if an unrecoverable error occurs. The low level functions abort by calling *longjmp* with a non-zero value.

If a low level function aborts, execution continues in *getall* as if its call to *setjmp* had just terminated with a non-zero value. Thus by testing the value returned by *setjmp* *getall* can determine whether *setjmp* is terminating because a low level function aborted.

```
#include "setjmp.h"

jmp__buf envbuf; /* environment saved here by setjmp */
getall(ptr)
char *ptr; /* ptr to record to be built */
{
    if (setjmp(envbuf))
        /* a low level function has aborted */
        return 0;
    getfield1(ptr);
    getfield2(ptr);
    getfield3(ptr);
    return ptr;
}
Here's one of the low level functions:
getsubfld21(ptr)
char *ptr;
{
    ...
    if (error)
        longjmp(envbuf, -1);
    ...
}
```


NAME

trigonometric functions:

sin, cos, tan, cotan, asin, acos, atan, atan2

SYNOPSIS

```
#include <math.h>
```

```
double sin(x)
```

```
double x;
```

```
double cos(x)
```

```
double x;
```

```
double tan(x)
```

```
double x;
```

```
double cotan(x)
```

```
double x;
```

```
double asin(x)
```

```
double x;
```

```
double acos(x)
```

```
double x;
```

```
double atan(x)
```

```
double x;
```

```
double atan2(x,y)
```

```
double x;
```

DESCRIPTION

sin, *cos*, *tan*, and *cotan* return trigonometric functions of radian arguments.

asin returns the arc sin in the range $-\pi/2$ to $\pi/2$.

acos returns the arc cosine in the range 0 to π .

atan returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

atan2 returns the arc tangent of x/y in the range $-\pi$ to π .

SEE ALSO

Errors (O)

DIAGNOSTICS

If a trig function can't perform the computation, it returns an arbitrary value and sets a code in the global integer *errno*; otherwise, it returns the computed number, without modifying *errno*.

A function will return the symbolic value EDOM if the argument is invalid, and the value ERANGE if the function value can't be computed. EDOM and ERANGE are defined in the file *errno.h*.

The values returned by the trig functions when the computation can't be performed are listed below. The symbolic values are defined in *math.h*.

function	error	f(x)	meaning
sin	ERANGE	0.0	$\text{abs}(x) > \text{XMAX}$
cos	ERANGE	0.0	$\text{abs}(x) > \text{XMAX}$
tan	ERANGE	0.0	$\text{abs}(x) > \text{XMAX}$
cotan	ERANGE	HUGE	$0 < x < \text{XMIN}$
cotan	ERANGE	-HUGEi	$-\text{XMIN} < x < 0$
cotan	ERANGE	0.0	$\text{abs}(x) \geq \text{XMAX}$
asin	EDOM	0.0	$\text{abs}(x) > 1.0$
acos	EDOM	0.0	$\text{abs}(x) > 1.0$
atan2	EDOM	0.0	$x = y = 0$

NAME

sinh, cosh, tanh

SYNOPSIS

```
#include <math.h>
```

```
double sinh(x)
```

```
double x;
```

```
double cosh(x)
```

```
double x;
```

```
double tanh(x)
```

```
double x;
```

DESCRIPTION

These functions compute the hyperbolic functions of their arguments.

SEE ALSO

Errors (O)

DIAGNOSTICS

If the absolute value of the argument to *sinh* or *cosh* is greater than 348.6, the function sets the symbolic value ERANGE in the global integer *errno* and returns a huge value. This code is defined in the file *errno.h*.

If no error occurs, the function returns the computed value without modifying *errno*.

NAME

`strcat`, `strncat`, `strcmp`, `strncmp`, `strcpy`, `strncpy`,
`strlen`, `index`, `rindex` - string operations

SYNOPSIS

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings, as follows:

strcat appends a copy of string *s2* to string *s1*. *strncat* copies at most *n* characters. Both terminate the resulting string with the null character (`\0`) and return a pointer to the first character of the resulting string.

strcmp compares its two arguments and returns an integer greater than, equal, or less than zero, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but looks at *n* characters at most.

strcpy copies string *s2* to *s1* stopping after the null character has been moved. *strncpy* copies exactly *n* characters: if *s2* contains less than *n* characters, null characters will be appended to the resulting string until *n* characters have been moved; if *s2* contains *n* or more characters, only the first *n* will be moved, and the resulting string will not be null terminated.

strlen returns the number of characters which occur in *s* up to the first null character.

index returns a pointer to the first occurrence of the character *c* in string *s*, or zero if *c* isn't in the string.

rindex returns a pointer to the last occurrence of the character *c* in string *s*, or zero if *c* isn't in the string.

NAME

toupper, tolower

SYNOPSIS

toupper(c)

tolower(c)

#include "ctype.h"

__toupper(c)

__tolower(c)

DESCRIPTION

toupper converts a lower case character to upper case: if *c* is a lower case character, *toupper* returns its upper case equivalent as its value, otherwise *c* is returned.

tolower converts an upper case character to lower case: if *c* is an upper case character *tolower* returns its lower case equivalent, otherwise *c* is returned.

toupper and *tolower* do not require the header file *ctype.h*.

__toupper and *__tolower* are macro versions of *toupper* and *tolower*, respectively. They are defined in *ctype.h*. The difference between the two sets of functions is that the macro versions will sometimes translate non-alphabetic characters, whereas the function versions don't.

NAME

`ungetc` - push a character back into input stream

SYNOPSIS

```
#include "stdio.h"
```

```
ungetc(c, stream)
```

```
FILE *stream;
```

DESCRIPTION

ungetc pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *ungetc* returns *c* as its value.

Only one character of pushback is guaranteed. EOF cannot be pushed back.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

ungetc returns EOF (-1) if the character can't be pushed back.

NAME

unlink

SYNOPSIS

```
unlink(name)
char *name;
```

DESCRIPTION

unlink erases a file.

name is a pointer to a character array containing the name of the file to be erased.

unlink returns 0 if successful.

DIAGNOSTICS

unlink returns -1 if it couldn't erase the file and places a code in the global integer *errno* describing the error.

NAME

write

SYNOPSIS

```
write(fd,buf,bufsize)  
int fd, bufsize; char *buf;
```

DESCRIPTION

write writes characters to a device or disk which has been previously opened by a call to *open* or *creat*. The characters are written to the device or file directly from the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer containing the characters to be written.

bufsize is the number of characters to be written.

If the operation is successful, *write* returns as its value the number of characters written.

SEE ALSO

Unbuffered I/O (O) , open, close, read

DIAGNOSTICS

If the operation is unsuccessful, *write* returns -1 and places a code in the global integer *errno*.

STYLE

Chapter Contents

Style style

1. Introduction 3

2. Structured Programming 7

3. Top-down Programming 8

4. Defensive Programming and Debugging 10

5. Things to watch out for 15

Style

This section was written for the programmer who is new to the C language, to communicate the special character of C and the programming practices for which it is best suited. This material will ease the new user's entry into C. It gives meaning to the peculiarities of C syntax, in order to avoid the errors which will otherwise disappear only with experience.

1. Introduction

what's in it for me?

These are the benefits to be reaped by following the methods presented here:

- * Reduced debugging times;
- * Increased program efficiency;
- * Reduced software maintenance burden.

The aim of the responsible programmer is to write straightforward code, which makes his programs more accessible to others. This section on style is meant to point out which programming habits are conducive to successful C programs and which are especially prone to cause trouble.

The many advantages of C can be abused. Since C is a terse, subtle language, it is easy to write code which is unclear. This is contrary to the "philosophy" of C and other structured programming languages, according to which the structure of a program should be clearly defined and easily recognizable.

keep it simple

There are several elements of programming style which make C easier to use. One of these is *simplicity*. Simplicity means *keep it simple*. You should be able to see exactly what your code will do, so that when it doesn't you can figure out why.

A little suspicion can also be useful. The particular "problem areas" which are discussed later in this section are points to check when code "looks right" but does not work. A small omission can cause many errors.

learn the C idioms

C becomes more valuable and more flexible with time. Obviously, elementary problems with syntax will disappear. But more importantly,

C can be described as "idiomatic." This means that certain expressions become part of a standard vocabulary used over and over.

For example,

```
while ((c = getchar()) != EOF)
```

is readily recognized and written by any C programmer. This is often used as the beginning of a loop which gets a character at a time from a source of input. Moreover, the inside set of parentheses, often omitted by a new C programmer, is rarely forgotten after this construct has been used a few times.

be flexible in using the library

The standard library contains a choice of functions for performing the same task. Certain combinations offer advantages, so that they are used routinely. For instance, the standard library contains a function, *scanf*, which can be used to input data of a given format. In this example, the function "scans" input for a floating point number:

```
scanf("%f", &flt_num);
```

There are several disadvantages to this function. An important debit is that it requires a lot of code. Also, it is not always clear how this function handles certain strings of input. Much time could be spent researching the behavior of this function. However, the equivalent to the above is done by the following:

```
flt_num = atof(gets(inp_buf));
```

This requires considerably less code, and is somewhat more straightforward. *gets* puts a line of input into the buffer, "inp_buf," and *atof* converts it to a floating point value. There is no question about what the input function is "looking for" and what it should find.

Furthermore, there is greater flexibility in the second method of getting input. For instance, if the user of the program could enter either a special command ("e" for exit) or a floating point value, the following is possible:

```
gets(inp_buf);
if (inp_buf[0] == 'e')
    exit(0);
flt_num = atof(inp_buf);
```

Here, the first character of input is checked for an "e", before the input is converted to a float.

The relative length of the library description of the *scanf* function is an indication of the problems that can arise with that and related functions.

write readable code

Readability can be greatly enhanced by adhering to what common sense dictates. For instance, most lines can easily accommodate more than one statement. Although the compiler will accept statements which are packed together indiscriminately, the logic behind the code will be lost. Therefore, it makes sense to write no more than one statement per line.

In a similar vein, it is desirable to be generous with whitespace. A blank space should separate the arithmetic and assignment operators from other symbols, such as variable names. And when parentheses are nested, dividing them with spaces is not being too prudent. For example,

```
if((fp=fopen("filename","r")==NULL))
```

is not the same as

```
if ( (fp = fopen("filename", "r")) == NULL )
```

The first line contains a misplaced parenthesis which changes the meaning of the statement entirely. (A file is opened but the file pointer will be null.) If the statement was expanded, as in the second line, the problem could be easily spotted, if not avoided altogether.

use straightforward logical expressions

Conditionals are apt to grow into long expressions. They should be kept short. Conditionals which extend into the next line should be divided so that the logic of the statement can be visualized at a glance. Another solution might be to reconsider the logic of the code itself.

learn the rules for expression evaluation

Keep in mind that the evaluation of an expression depends upon the order in which the operators are evaluated. This is determined from their relative precedence.

Item 7 in the list of "things to watch out for", below, gives an example of what may happen when the evaluation of a boolean expression stops "in the middle". The rule in C is that a boolean will be evaluated only until the value of the expression can be determined.

Item 8 gives a well known example of an "undefined" expression, one whose value is not strictly determined.

In general, if an expression depends upon the order in which it is evaluated, the results may be dubious. Though the result may be strictly defined, you must be certain you know what that definition is.

a matter of taste

There are several popular styles of indentation and placement of the braces enclosing compound statements. Whichever format you

adopt, it is important to be consistent. Indentation is the accepted way of conveying the intended nesting of program statements to other programmers. However, the compiler understands only braces. Making them as visible as possible will help in tracking down nesting errors later.

However much time is devoted to writing readable code, C is low-level enough to permit some very peculiar expressions.

`/* It is important to insert comments on a regular basis! */`

Comments are especially useful as brief introductions to function definitions.

In general, moderate observance of these suggestions will lessen the number of "tricks" C will play on you-- even after you have mastered its syntax.

2. Structured Programming

"Structured programming" is an attempt to encourage programming characterized by method and clarity. It stems from the theory that any programming task can be broken into simpler components. The three basic parts are statements, loops, and conditionals. In C, these parts are, respectively, anything enclosed by braces or ending with a semicolon; *for*, *while* and *do-while*; *if-else*.

modularity and block structure

Central to structured programming is the concept of modularity. In one sense, any source file compiled by itself is a module. However, the term is used here with a more specific meaning. In this context, modularity refers to the independence or isolation of one routine from another. For example, a routine such as *main()* can call a function to do a given task even though it does not know how the task is accomplished or what intermediate values are used to reach the final result.

Sections of a program set aside by braces are called "blocks". The "privacy" of C's block structure ensures that the variables of each block are not inadvertently shared by other blocks. Any left brace ({} signals the beginning of a block, such as the body of a function or a *for* loop. Since each block can have its own set of variables, a left brace marks an opportunity to declare a temporary variable.

A function in C is a special block because it is called and is passed control of execution. A function is called, executes and returns. Essentially, a C program is just such a routine, namely, *main*.

A function call represents a task to be accomplished. Program statements which might otherwise appear as several obscure lines can be set aside in a function which satisfies a desired purpose. For instance, *getchar* is used to get a single character from standard input.

When a section of code must be modified, it is simpler to replace a single modular block than it is to delete a section of an unstructured program whose boundaries may be unclear at best. In general, the more precisely a block of program is defined, the more easily it can be changed.

3. Top-down Programming

"Top-down" programming is one method that takes advantage of structured programming features like those discussed above. It is a method of designing, writing, and testing a program from the most general function (i.e., `(main())`) to the most specific functions (such as `getchar()`).

All C programs begin with a function called `main()`. `main()` can be thought of as a supervisor or manager which calls upon other functions to perform specific tasks, doing little of the work itself. If the overall goal of the program can be considered in four parts (for instance, input, processing, error checking and output), then `main()` should call at least four other functions.

step one

The first step in the design of a program is to identify what is to be done and how it can be accomplished in a "programmable" way. The *main* routine should be greatly simplified. It needs to call a function to perform the crucial steps in the program. For example, it may call a function, `init()`, which takes care of all necessary startup initializations. At this point, the programmer does not even need to be certain of all the initializations that will take place in `init()`.

All functions consist of three parts: a parameter list, body, and return value. The design of a function must focus on each of these three elements.

During this first stage of design, each function can be considered a black box. We are concerned only with what goes in and what comes out, not with what goes on inside.

Do not allow yourself to be distracted by the details of the implementation at this point. Flowcharts, pseudocode, decision tables and the like are useful at this stage of the implementation.

A detailed list of the data which is passed back and forth between functions is important and should not be neglected. The interface between functions is crucial.

Although all functions are written with a purpose in mind, it is easy to unwittingly merge two tasks into one. Sometimes, this may be done in the interests of producing a compact and efficient program function. However, the usual result is a bulky, unmanageable function. If a function grows very large or if its logic becomes difficult to comprehend, it should be reduced by introducing additional function calls.

step two

There comes a time when a program must pass from the design stage into the coding stage. You may find the top-down approach to

coding too restrictive. According to this scheme, the smallest and most specific functions would be coded last. It is our nature to tackle the most daunting problems first, which usually means coding the low-level functions.

Whereas the top-down approach is the preferred method for designing software, the bottom-up approach is often the most practical method for writing software. Given a good design, either method of implementation should produce equally good results.

One asset of top-down writing is the ability to provide immediate tests on upper level routines. Unresolved function calls can be satisfied by "dummy" functions which return a range of test values. When new functions are added, they can operate in an environment that has already been tested.

C functions are most effective when they are as mutually independent as is possible. This independence is encouraged by the fact that there is normally only one way into and one way out of a function: by calling it with specific arguments and returning a meaningful value. Any function can be modified or replaced so long as its entry and exit points are consistent with the calling function.

4. Defensive Programming and Debugging

"Defensive programming" obeys the same edict as defensive driving: trust no one to do what you expect. There are two sides to this rule of thumb. Defend against both the possibility of bad data or misuse of the program by the user, and the possibility of bad data generated by bad code.

Pointers, for example, are a prime source of variables gone astray. Even though the "theory" of pointers may be well understood, using them in new ways (or for the first time) requires careful consideration at each step. Pointers present the fewest problems when they appear in familiar settings.

faced with the unknown

When trying something new, first write a few test programs to make sure the syntax you are using is correct. For example, consider a buffer, `str_buf`, filled with null-terminated strings. Suppose we want to print the string which begins at offset *begin* in the buffer. Is this the way to do it?

```
printf("%s", str_buf[begin]);
```

A little investigation shows that `str_buf[begin]` is a character, not a pointer to a string, which is what is called for. The correct statement is

```
printf("%s", str_buf + begin);
```

This kind of error may not be obvious when you first see it. There are other topics which can be troublesome at first exposure. The promotion of data types within expressions is an example. Even if you are sure how a new construct behaves, it never hurts to doublecheck with a test program.

Certain programming habits will ease the bite of syntax. Foremost among these is simplicity of style. Top-down programming is aimed at producing brief and consequently simple functions. This simplicity should not disappear when the design is coded.

Code should appear as "idiomatic" as possible. Pointers can again provide an example: it is a fact of C syntax that arrays and pointers are one and the same. That is,

```
array[offset]
```

is the same as

```
*(array + offset)
```

The only difference is that an array name is not an lvalue; it is fixed. But mixing the two ways of referencing an object can cause confusion, such as in the last example. Choosing a certain idiom, which is known to behave a certain way, can help avoid many errors in usage.

when bugs strike

The assumption must be that you will have to return to the source code to make changes, probably due to what is called a bug. Bugs are characterized by their persistence and their tendency to multiply rapidly.

Errors can occur at either compile-time or run-time. Compile-time errors are somewhat easier to resolve since they are usually errors in syntax which the compiler will point out.

from the compiler

If the compiler does pick up an error in the source code, it will send an error code to the screen and try to specify where the error occurred. There are several peculiarities about error reporting which should be brought up right away.

The most noticeable of these peculiarities is the number of spurious errors which the compiler may report. This interval of inconsistency is referred to as the compiler's recovery. The safest way to deal with an unusually long list of errors is to correct the first error and then recompile before proceeding.

The compiler will specify where it "noticed" something was wrong. This does not necessarily indicate where you must make a change in the code. The error number is a more accurate clue, since it shows what the compiler was looking for when the error occurred.

if this ever happens to you

A common example of this is error 69: "missing semicolon." This error code will be put out if the compiler is expecting a semicolon when it finds some other character. Since this error most often occurs at the end of a line, it may not be reported until the first character of the following line-- recall that whitespace, such as a newline character, is ignored.

Such an error can be especially treacherous in certain situations. For example, a missing semicolon at the end of a *#include*'d file may be reported when the compiler returns to read input in the original file.

In general, it is helpful to look at a syntax error from the compiler's point of view.

Consider this error:

```
struct structag {  
    char c;  
    int i;  
}  
  
int j;
```

This should generate an error 16: "data type conflict". The arrow in the error message should show that the error was detected right after the "int" in the declaration of *j*. This means that the error has to do with something before that line, since there is nothing illegal about the *int* keyword.

By inspection, we may see that the semicolon is missing from the preceding line. If this fact escapes our notice, we still know that error 16 means this: the compiler found a declaration of the form

[data type] [data type] [symbol name]

where the two data types were incompatible. So while *shortint* is a good data type, *double int* is not. A small intuitive leap leads us to assume that the compiler has read our source as a kind of "struct int" declaration; *struct* is the only keyword preceding the *int* which could have caused this error. Since the compiler is reading the two declarations as a single statement, we must be missing a delimiter.

run-time errors

It takes a bit more ingenuity to locate errors which occur at run-time. In numerical calculations, only the most anomalous results will draw attention to themselves. Other bugs will generate output which will appear to have come from an entirely different program.

A bug is most useful when it is repeatable. Bugs which show up only "sometimes" are merely vexing. They can be caused by a corrupted disk file or a bad command from the user.

When an error can be consistently produced, its source can be more easily located. The nature of an error is a good clue as to its source. Much of your time and sanity will be preserved by setting aside a few minutes to reflect upon the problem.

Which modules are involved in the computation or process? Many possibilities can be eliminated from the start, such as pieces of code which are unrelated to the error.

The first goal is to determine, from a number of possibilities, which module might be the source of the bug.

checking input data

Input to the program can be checked at a low cost. Error checking of this sort should be included on a "routine" basis. For instance, "if ((fp=fopen("file","r"))==NULL)" should be reflex when a file is

opened. Any useful error handling can follow in the body of the *if*.

It is easy to check your data when you first get your hands on it. If an error occurs after that, you have a bug in your program.

printf it

It is useful to know where the data goes awry. One brute force way of tracking down the bug is to insert *printf* statements wherever the data is referenced. When an unexpected value comes up, a single module can be chosen for further investigation.

The *printf* search will be most effective when done with more refinement. Choose a suspect module. There are only two keys points to check: the entry and return of the function. *printf* the data in question just as soon as the function is entered. If the values are already incorrect, then you will want to make sure the correct data was passed in the function call.

If an incorrect value is returned, then the search is confined to the guilty function. Even if the function returns a good value, you may want to make sure it is handled correctly by the calling function.

If everything seems to be working, jump to the next tricky module and perform another check. When you find a bad result, you will still have to backtrack to discover precisely where the data was spoiled.

function calls

Be aware that data can be garbled in a function call. Function parameters must be declared when they are not two byte integers. For instance, if a function is called:

```
fseek(fp, 0, 0);
```

in order to "seek" to the beginning of a file, but the function is defined this way:

```
fseek(fp, offset, origin)
FILE *fp;
long offset;
int origin;
```

there will be unfortunate consequences.

The second parameter is expected to be a *long* integer (four bytes), but what is being passed is a *short* integer (two bytes). In a function call, the arguments are just being pushed onto the stack; when the function is entered, they are pulled off again. In the example, two bytes are being pushed on, but four bytes (whatever four bytes are there) are being pulled off.

The solution is just to make the second parameter a long, with a suffix (0L) or by the cast operator (as in (long)i).

A similar problem occurs when a non-integer return value is not declared in the calling function. For example, if *sqrt* is being called, it must be declared as returning a *double*:

```
double sqrt();
```

This method of debugging demonstrates the usefulness of having a solid design before a function is coded. If you know what should be going into a function and what should be coming out, the process of checking that data is made much simpler.

found it

When the guilty function is isolated, the difficulty of finding the bug is proportional to the simplicity of the code. However, the search can continue in a similar way. You should have a good notion of the purpose of each block, such as a loop. By inserting a *printf* in a loop, you can observe the effect of each pass on the data.

printf's can also point out which blocks are actually being executed. "Falling through" a test, such as an *if* or a *switch*, can be a subtle source of problems. Conditionals should not leave cases untested. An *else*, or a *default* in a *switch*, can rescue the code from unexpected input.

And if you are uncertain how a piece of code will work, it is usually worthwhile to set up small test programs and observe what happens. This is instructional and may reveal a bug or two.

5. Things to Watch Out for

Some errors arise again and again. Not all of them go away with experience. The following list will give you an idea of the kinds of things that can go wrong.

- * **missing semicolon or brace**

The compiler will tell you when a missing semicolon or brace has introduced bad syntax into the code. However, often such an error will affect only the logical structure of the program; the code may compile and even execute. When this error is not revealed by inspection, it is usually brought out by a test *printf* which is executed too often or not enough. See compiler error 69.

- * **assignment (=) vs comparison (==)**

Since variables are assigned values more often than they are tested for equality, the former operator was given the single keystroke: =. Notice that all the comparison tests with equality are two characters: <=, >= and ==.

- * **misplaced semicolon**

When typing in a program, keep in mind that all source lines do not automatically end with a semicolon. Control lines are especially susceptible to an unwanted semicolon:

```
for (i=0; i<100; i++);  
    printf("%d",i);
```

This example prints the single number 100.

- * **division (/) vs escape sequence (\)**

C definitely distinguishes between these characters. The division sign resides below the question mark on a standard console; the backslash is generally harder to find.

- * **character constant vs character string**

Character constants are actually integers equal to the ASCII values of the respective character. A character string is a series of characters terminated by a null character (\0). The appropriate delimiter is the single quote and double quote, respectively.

- * **uninitialized variable**

At some point, all variables must be given values before they are used. The compiler will set global and static variables to zero, but automatic variables are guaranteed to contain garbage every time they are created.

* evaluation of expressions

For most operations in C, the order of evaluation is rigidly defined; thus, many expressions can be written without lots of parentheses.

However, the order in which unparenthesized expressions are evaluated are not always what you would expect; therefore, it's usually a good idea to use parentheses liberally in expressions where there may be doubt about the order of evaluation (in your mind or in the mind of someone who may later read your program).

For example, the result of the following example is 6:

```
int a = 2, b = 3, c = 4, d;
d = a + b / a * c;
```

The above expression is equivalent to the parenthesized expression $d = a + ((b / a) * c)$. You should probably use some parentheses in this expression, to make its effect clear to yourself and to others.

Consider this example:

```
if ( (c = 0) || (c = 1) )
    printf("%d", c);
```

"1" will be printed; since the first half of the conditional evaluates to zero, the second half must be also evaluated. But in this example:

```
if ( (c = 0) && (c = 1) )
    ;
printf("%d", c);
```

a "0" is printed. Since the first half evaluates to zero, the value of the conditional must be zero, or false, and evaluation stops. This is a property of the logical operators.

* undefined order of evaluation

Unfortunately, not all operators were given a complete set of instructions as to how they should be evaluated. A good example is the increment (or decrement) operator. For instance, the following is undefined:

```
i = ++i + --i / ++i - i++;
```

How such an expression is evaluated by a particular implementation is called a "side effect." In general, side effects are to be avoided.

* evaluation of boolean expressions

Ands, ors and nots invite the programmer to write long conditionals whose very purpose is lost in the code. Booleans should be brief and to the point. Also, the bitwise logical operators must be fully parenthesized. The table in sections 2.12 and 18.1 of *The C Programming Language*, by Kernighan and Ritchie, shows their precedence in relation to other operators.

Here is an extreme example of how a lengthy boolean can be reduced:

```
if ((c = getchar()) != EOF && c >= 'a' && c <= 'z' &&
    (c = getchar()) >= '1' && c <= '9')
    printf("good input\n");

if ((c = getchar()) != EOF)
    if (c >= 'a' && c <= 'z')
        if ((c = getchar()) >= '0' && c <= '9')
            printf("good input\n");
```

* badly formed comments

The theory of comment syntax is simply that everything occurring between a left `/*` and a right `*/` is ignored by the compiler. Nonetheless, a missing `*/` should not be overlooked as a possible error.

Note that comments cannot be nested, that is

```
/* /* this will cause an error */ */
```

And this could happen to you too:

```
/* the rest of this file is ignored until another comment /*
```

* nesting error

Remember that nesting is determined by braces and not by indentations in the text of the source. Nested *if* statements merit particular care since they are often paired with an *else*.

* usage of else

Every *else* must pair up with an *if*. When an *else* has inexplicably remained unpaired, the cause is often related to the first error in this list.

* falling through the cases in a switch

To maintain the most control over the *cases* in a *switch* statement, it is advisable to end each *case* with a *break*, including the last *case* in the *switch*.

* strange loops

The behavior of loops can be explored by inserting *printf* statements in the body of the loop. Obviously, this will indicate if the loop has even been entered at all in course of a run. A counter will show just how many times the loop was executed; a small slip-up will cause a loop to be run through once too often or seldom. The condition for leaving the loop should be doublechecked for accuracy.

- * **use of strings**

All strings must be terminated by a null character in memory. Thus, the string, "hello", will occupy a six-element array; the sixth element is ' '. This convention is essential when passing a string to a standard library function. The compiler will append the null character to string constants automatically.

- * **pointer vs object of a pointer**

The greatest difficulty in using pointers is being sure of what is needed and what is being used. Functions which take a pointer argument require an address in memory. The best way to ensure that the correct value is being passed is to keep track of what is being pointed to by which pointer.

- * **array subscripting**

The first element in a C array has a subscript of zero. The array name without a subscript is actually a pointer to this element. Obviously, many problems can develop from an incorrect subscript. The most damaging can be subscripting out of bounds, since this will access memory above the array and overwrite any data there. If array elements or data stored with arrays are being lost, this error is a good candidate.

- * **function interface**

During the design stage, the components of a program should be associated with functions. It is important that the data which is passed among or shared by these functions be explicitly defined in the preliminary design of the program. This will greatly facilitate the coding of the program since the interface between functions must be precise in several respects.

First of all, if the parameters of a function are established, a call can be made without the reservation that it will be changed later. There is less chance that the arguments will be of the wrong type or specified in the wrong order.

A function is given only a private copy of the variables it is passed. This is a good reason to decide while designing the program how functions should access the data they require. You will be able to detail the arguments to be passed in a function call, the global data which the function will alter, the value which the function will return and what declarations will be appropriate-- all without concern for how the function will be coded.

Argument declarations should be a fairly simple matter once these things are known. Note that this declaration list must stand before the left brace of the function body.

The type of the function is the same as the type of the value it returns. Functions must be declared just like any variable. And just like variables, functions will default to type int, that is, the compiler will assume that a function returns an integer if you do not tell it otherwise with a declaration. Thus if function f calls function g which returns a variable of type double, the following declaration is needed:

```
function f()
{
    double g(), bigfloat;

    g(bigfloat);
}
double g(arg)
double arg;
{
    return(arg);
}
```

*** be sure of what a function returns**

You will probably know very well what is returned by a function you have written yourself. But care should be taken when using functions coded by someone else. This is especially true of the standard library functions. Most of the supplied library functions will return an int or a char pointer where you might expect a char. For instance, getchar() returns an int, not a char. The functions supplied by Manx adhere to the UNIX model in all but a few cases.

Of course, the above applies to a function's arguments as well.

*** shared data**

Variables that are declared globally can be accessed by all functions in the file. This is not a very safe way to pass data to functions since once a global variable is altered, there is no returning it to its former state without an elaborate method of saving data. Moreover, global data must be carefully managed; a function may process the wrong variable and consequently inhibit any other function which depends on that data.

Since C provides for and even encourages private data, this definitely should not be a common bug.

COMPILER ERROR MESSAGES

Chapter Contents

Compiler Error Codes err

1. Summary 4

2. Explanations 7

3. Fatal Error Messages 35

Compiler Error Messages

This chapter discusses error messages that can be generated by the compiler. It is divided into three sections: the first summarizes the messages, the second explains them, and the third discusses fatal compiler error messages.

1. Summary of error codes

No. Interpretation

- 1: bad digit in octal constant
- 2: string space exhausted
- 3: unterminated string
- 4: internal error
- 5: illegal type for function
- 6: inappropriate arguments
- 7: bad declaration syntax
- 8: syntax error in typecast
- 9: array dimension must be constant
- 10: array size must be positive integer
- 11: data type too complex
- 12: illegal pointer reference
- 13: unimplemented type
- 14: internal
- 15: internal
- 16: data type conflict
- 17: unsupported data type
- 18: data type conflict
- 19: obsolete
- 20: structure redeclaration
- 21: missing }
- 22: syntax error in structure declaration
- 23: incorrect type for library function (Apprentice C only)
obsolete (other Aztec C compilers)
- 24: need right parenthesis or comma in arg list
- 25: structure member name expected here
- 26: must be structure/union member
- 27: illegal typecast
- 28: incompatible structures
- 29: illegal use of structure
- 30: missing : in ? conditional expression
- 31: call of non-function
- 32: illegal pointer calculation
- 33: illegal type
- 34: undefined symbol
- 35: typedef not allowed here
- 36: no more expression space
- 37: invalid expression for unary operator
- 38: no auto. aggregate initialization allowed
- 39: obsolete
- 40: internal
- 41: initializer not a constant
- 42: too many initializers

43: initialization of undefined structure
44: obsolete
45: bad declaration syntax
46: missing closing brace
47: open failure on include file
48: illegal symbol name
49: multiply defined symbol
50: missing bracket
51: lvalue required
52: obsolete
53: multiply defined label
54: too many labels
55: missing quote
56: missing apostrophe
57: line too long
58: illegal # encountered
59: macro too long
60: obsolete
61: reference of member of undefined structure
62: function body must be compound statement
63: undefined label
64: inappropriate arguments
65: illegal argument name
66: expected comma
67: invalid else
68: syntax error
69: missing semicolon
70: goto needs a label
71: statement syntax error in do-while
72: 'for' syntax error: missing first semicolon
73: 'for' syntax error: missing second semicolon
74: case value must be an integer constant
75: missing colon on case
76: too many cases in switch
77: case outside of switch
78: missing colon on default
79: duplicate default
80: default outside of switch
81: break/continue error
82: illegal character
83: too many nested includes
84: too many array dimensions
85: not an argument
86: null dimension in array
87: invalid character constant
88: not a structure
89: invalid use of register storage class
90: symbol redeclared

- 91: illegal use of floating point type
- 92: illegal type conversion
- 93: illegal expression type for switch
- 94: invalid identifier in macro definition
- 95: macro needs argument list
- 96: missing argument to macro
- 97: obsolete
- 98: not enough arguments in macro reference
- 99: internal
- 100: internal
- 101: missing close parenthesis on macro reference
- 102: macro arguments too long
- 103: #else with no #if
- 104: #endif with no #if
- 105: #endasm with no #asm
- 106: #asm within #asm block
- 107: missing #endif
- 108: missing #endasm
- 109: #if value must be integer constant
- 110: invalid use of : operator
- 111: invalid use of void expression
- 112: invalid use function pointer
- 113: duplicate case in switch
- 114: macro redefined
- 115: keyword redefined
- 116: field width must be > 0
- 117: invalid 0 length field
- 118: field is too wide
- 119: field not allowed here
- 120: invalid type for field
- 121: ptr to int conversion
- 122: ptr & int not same size
- 123: function ptr & ptr not same size
- 124: invalid ptr/ptr assignment
- 125: too many subscripts or indirection on integer

Error codes between 116 and 125 will not occur on Aztec C compilers whose version number is less than 3.

Error codes greater than 200 will occur only if there's something wrong with the compiler. If you get such an error, please send us the program that generated the error.

2. Explanations

1: bad digit in octal constant

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2: string space exhausted

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option `-Z`. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program.

3: unterminated string

All strings must begin and end with double quotes (`"`). This message indicates that a double quote has remained unpaired.

4: internal error

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of MANX. It could be a bug in the compiler. The release documentation enclosed with the product contains further information.

5: illegal type for function

The type of a function refers to the type of the value which it returns. Functions return an *int* by default unless they are declared otherwise. However, functions are not allowed to return aggregates (arrays or structures). An attempt to write a function such as *struct sam func()* will generate this error code. The legal function types are *char*, *int*, *float*, *double*, *unsigned*, *long*, *void* and a pointer to any type (including structures).

6: error in argument declaration

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to *int*, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
shrt arg 1; /* misspelled or invalid keyword */
double arg 2;
{ /* function body */
}

goodfunction(arg1,arg2)
float arg1;
int arg2; /* this line is not required */
{ /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;          /* correct */
char c d;          /* error 7 */
char *s1, *s2
float k;           /* error 7 detected here */
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a *#include*'d file will be detected back in the file being compiled or in another *#include* file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in type cast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number); /* incorrect usage */
i = 3 * ((int)number); /* correct usage */
```

9: array dimension must be constant

The dimension given an array must be a constant of type *char*, *int*, or *unsigned*. This value is specified in the declaration of the array. See error 10.

10: array size must be positive integer

The dimension of an array is required to be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, specifying a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];           /* meaningless */
extern char goodarray[];    /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, *goodarray* is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
{
    ...
}
```

11: data type too complex

This message is best explained by example:

```
char *****foo;
```

The form of this declaration implies six pointers-to-pointers. The seventh asterisk indicates a pointer to a *char*. The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error; all that is being declared in any case is a single two-byte pointer. However it is to be hoped that such a construct will never be needed.

12: illegal pointer reference

The type of a pointer must be either *int* or *unsigned*. This is why you might get away with not declaring pointer arguments in functions like *fopen* which return a pointer; they default to *int*. When this error is generated, an expression used as a pointer is of an invalid type:

```
char c;
int var;                               /* any variable */
int varaddress;
varaddress = &var;                     /* valid since addresses */
*(varaddress) = 'c';                   /* can fit in an int */
*(expression) = 10;                   /* in general, expression
                                       must be an int or unsigned */
*c = 'c';                             /* error 12 */
```

13: internal [see error 4]

14: internal [see error 4]

15: storage class conflict

Only automatic variables and function parameters can be specified as *register*.

This error can be caused by declaring a *static register* variable. While structure members cannot be given a storage class at all, function

arguments can be specified only as *register*.

A *register int i* declaration is not allowed outside a function--it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say *long int i*, and *unsigned int j*, it is meaningless to use *double int k* or *float char c*. In this respect, the compiler checks to make sure that *int*, *char*, *float* and *double* are used correctly.

<i>data type</i>	<i>interpretation</i>	<i>size(bytes)</i>
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
long/long integer	long integer	4
float	floating point number	4
long float/double	double precision float	8

17: Unsupported data type

This message occurs only when data types are used which are supported by the extended C language, such as the *enum* data type.

18: data type conflict

This message indicates an error in the use of the *long* or *unsigned* data type. *long* can be applied as a qualifier to *int* and *float*. *unsigned* can be used with *char*, *int* and *long*.

```
long i;                /* a long int */
long float d;          /* a double */
unsigned u;            /* an unsigned int */
unsigned char c;
unsigned long l;
unsigned float f;      /* error 18 */
```

19: obsolete

Error codes interpreted as obsolete do not occur in the current version of the compiler. Some simply no longer apply due to the increased adaptability of the compiler. Other error codes have been translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact Manx for information.

20: structure redeclaration

The compiler is able to tell you if a *structure* has already been defined. This message informs you that you have tried to redefine a *structure*.

21: missing }

The compiler expects to find a comma after each member in the list of fields for a *structure* initialization. After the last field, it expects a right (close) brace.

For example, the following program fragment will generate error 21, since the initialization of the structure named 'harry' doesn't have a closing brace:

```
struct sam {  
    int bone;  
    char license[10];  
} harry = {  
    1,  
    "23-4-1984";
```

22: syntax error in structure declaration

The compiler was unable to find the left (open) brace which follows the tag in a *structure* declaration. In the example for error 21, "sam" is the structure tag. A left brace must follow the keyword *struct* if no structure tag is specified.

23: incorrect type for library function (Apprentice C only)

For Apprentice C, this error means that your program has either explicitly or implicitly incorrectly declared the type of a function that's in the run-time system. For example, you will get this error if you call the run-time system function *sqrt* without declaring that it returns a *double*.

23: obsolete (Other Aztec C Compilers)

For Compilers other than Apprentice C, this error should not occur.

24: need right parenthesis or comma

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that *getchar* is a function rather than a variable.

```
getchar();
```

This is the equivalent of

CALL getchar

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funcall(arg1, arg2 arg3);
```

25: structure member name expected here

The symbol name following the dot operator or the arrow must be valid. A valid name is a string of alphanumerics and underscores. It must begin with an alphabetic (a letter of the alphabet or an underscore). In the last line of the following example, "(salary)" is not valid because '(' is not an alphanumeric.

```
empptr = &anderson;
empptr->salary = 12000;      /* these three lines */
(*empptr).salary = 12000;   /* are */
anderson.salary = 12000;    /* equivalent */
empptr = &anderson.;       /* error 25 */
empptr-> = 12000;           /* error 25 */
anderson.(salary) = 12000; /* error 25 */
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the -S option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.

27: illegal type cast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure sam { ... } thom;
thom = (struct sam)(expression); /* error 27 */
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. Both structures must have the same structure tag. For example:

```
struct sam harry;
struct sam thom;

...
harry = thom;
```

29: illegal use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

```
expression ? statement1 : statement2
```

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

The following represents a function call:

```
symbol(arg1, arg2, ..., argn);
```

where "symbol" is not a reserved word and the expression stands in the body of a function. Error 31, in reference to the expression above, indicates that "symbol" has been previously declared as something other than a function.

A missing operator may also cause this error:

```
a(b + c);           /* error 31 */
a * (b + c);        /* intended */
```

The missing '*' makes the compiler view "a()" as a function call.

32: illegal pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. (For a formal definition, see Kernighan and Ritchie pp. 188-189.) Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: illegal type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct sam { ... } harry;
a = -array;           /* ? */
b = -harry;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of *sizeof(expression)* and the cast operator. Compare the accompanying examples:

```
struct sam {
    int i;
} harry;
typedef double bigfloat;
typedef struct sam foo;

j = 4 * bigfloat f;      /* error 35 */
k = &foo;                /* error 35 */
x = sizeof(bigfloat);
y = sizeof(foo);         /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as "harry"). It is no more meaningful to take the address of a structure type than any other data type, as in *&int*.

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the -E option to increase the number of available entries in the expression table. See the description of the compiler in the manual.

37: invalid expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and sizeof.

```
if (!) ;
```

38: no auto. aggregate initialization

It is not permitted to initialize automatic arrays and structures. Static and external aggregates may be initialized, but by default their members are set to zero.

```
char array[5] = { 'a', 'b', 'c', 'd' };
function()
{
    static struct sam {
        int bone;
        char license[10];
    } harry = {
        1,
        "123-4-1984"
    };
    char autoarray[2] = { 'f', 'g' }; /* no good */
    extern char array[];
}
```

There are three variables in the above example, only two of which are correctly initialized. The variable "array" may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.

The structure "harry" is static and may be initialized. Notice that "license" cannot be initialized without first giving a value to "bone". There are no provisions in C for setting a value in the middle of an aggregate.

The variable "autoarray" is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic aggregates cannot be initialized.

39: obsolete [see error 19]

40: internal [see error 4]

41: initializer not a constant

In certain initializations, the expression to the right of the equals sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actually occurs at link-time, and not at run-time.

```
{
    int i = 3;
    static int j = (2 + i);    /* illegal */
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
    struct {
        char array[];
    } substruct;
} superstruct =
```

version 1:

```
{
    "abcdefghij"
};
```

version 2:

```
{
    {
        { 'a','b','c',..., 'i','j' }
    }
};
```

In version 1, the initializers are copied byte-for-byte onto the structure, *superstruct*.

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[10] = "abcdefghij";
```

This will generate error 42 because the string constant on the right is null-terminated. The null terminator (' ' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43: undefined structure initialization

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct sam {...};  
struct dog sam = { 1, 2, 3}; /* error 43 */
```

44: obsolete [see error 19]

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the *while* loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the *while* loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```

main()
{
    int i, j;
    char array[80];
    gets(array);
    i = 0;
    while (array[i]) {
        putchar(array[i]);
        i++;
    }
    for (i=0; array[i]; i++) {
        for (j=i + 1; array[j]; j++) {
            printf("elements %d and %d are ", i, j);
            if (array[i] == array[j])
                printf("the same\n");
            else
                printf("different\n");
        }
        putchar('\n');
    }
}

```

47: open failure on include file

When a file is *#included*, the compiler will look for it in a default area (see the manual description of the compiler). This message will be generated if the file could not be opened. An open failure usually occurs when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in an include statement, but this diminishes flexibility somewhat.

48: illegal symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The succeeding characters may be any combination of alphanumeric (alphabets and numerals). The following symbols will produce this error code:

```

2nd__time,
dont__do__this!

```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```

int i, j, k, i;          /* illegal */

```


50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only *lvalues* are allowed to stand on the left-hand side of an assignment. For example:

```
int num;
num = 7;
```

They are distinguished from *rvalues*, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An *lvalue* may be thought of as a bucket into which an *rvalue* can be dropped. Just as the contents of one bucket can be passed to another, so can an *lvalue* *y* be assigned to another *lvalue*, *x*:

```
#define NUMBER 512
x = y;
1024 = z;           /* wrong; l/rvalues are reversed */
NUMBER = x;         /* wrong; NUMBER is still an rvalue */
```

Some operators which require *lvalues* as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
register int i, j;
i = 3;
j = &i;
```

52: obsolete [see error 19]**53: multiply defined label**

On occasions when the goto statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. Although this table is fixed in size, it should satisfy the requirements of any reasonable C program. C was structured to discourage extravagance in the use of goto's. Strictly speaking, goto statements are not required by any procedure in C; they are primarily recommended as a quick and simple means of exiting from a nested structure.

This error indicates that you should significantly reduce the number of goto's in your program.

55: missing quote

The compiler found a mismatched double quote (") in a *#define* preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"
```

```
"this is a string with an embedded quote: \"."
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a *#define* preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\';           /* c is initialized to  
                        single quote */
```

57: line too long

Lines are restricted in length by the size of the buffer used to hold them. This restriction varies from system to system. However, logical lines can be infinitely long by continuing a line with a backslash-newline sequence. These characters will be ignored.

58: illegal # encountered

The pound sign (#) begins each command for the preprocessor: *#include*, *#define*, *#if*, *#ifdef*, *#ifndef*, *#else*, *#endif*, *#asm*, *#endasm*, *#line* and *#undef*. These symbols are strictly defined. The pound sign (#) must be in column one and lower case letters are required.

59: macro too long

Macros can be defined with a preprocessor command of the following form:

```
#define [identifier] [substitution text]
```

The compiler then proceeds to replace all instances of "identifier" with the substitution text that was specified by the *#define*.

This error code refers to the substitution text of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (), for practical purposes the size of a macro has been limited to 255 characters.

60: obsolete [see error 19]

61: reference of member of undefined structure

Occurs only under compilation without the -S option. Consider the following example:

```
int bone;
struct cat {
    int toy;
} manx;
struct dog *sampr;
manx.toy = 1;
bone = sampr->toy;    /* error 61 */
```

This error code appears most often in conjunction with this kind of mistake. It is possible to define a pointer to a structure without having already defined the structure itself. In the example, *sampr* is a structure pointer, but what form that structure ("dog") may take is still unknown. So when reference is made to a member of the structure to which *sampr* points, the compiler replies that it does not even know what the structure looks like.

The -S compiler option is provided to duplicate the manner in which earlier versions of UNIX treated structures. Given the example above, it would make the compiler search all previously defined structures for the member in question. In particular, the value of the member "toy" found in the structure "manx" would be assigned to the variable "bone". The -S option is not recommended as a short cut for defining structures.

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()
{
    return 1;
}
```

This error can also be caused by an error inside a function declaration list, as in:

```
func(a, b)
int a; chr b;
{
    ...
}
```

63: undefined label

A *goto* statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to goto a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding goto, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function(string)
char *string;
{
    char *func1();           /* correct */
    double func2(x,y);      /* wrong */
    ...
}
```

In this example, function() is being defined, but func1() and func2() are being declared.

65: illegal or missing argument name

The compiler has found an illegal name in a function argument list. An argument name must conform to the same rules as variable names, beginning with an alphabetic (letter or underscore) and continuing with any sequence of alphanumerics and underscores. Names must not coincide with reserved words.

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An *else* was found which is not associated with an *if* statement. *else* is bound to the nearest *if* at its own level of nesting. So if-else pairings are determined by their relative placement in the code and their grouping by braces.

```
if(...) {
    ...
    if (...) {
        ...
    } else if (...)
        ...
    } else {
        ...
    }
}
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the if and else-if means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding if

statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the else statement. As shown here, the else is paired with the first if, not the second.

68: syntax error

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as *char* or *int* must not lead a statement; compare the use of the casting operator:

```
func()
{
    int i;
    char array[12];
    float k = 2.03;

    i = 0;
    int m;                      /* error 68 */
    j = i + 5;
    i = (int) k;                 /* correct */
    if (i) {
        int i = 3;
        j = i;
        printf("%d",i);
    }
    printf("%d%d\n",i,j);
}
```

This trivial function prints the values 3, 2 and 3. The variable *i* which is declared in the body of the conditional (if) lives only until the next right brace; then it dies, and the original *i* regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is subject to the same vagaries as its cousin, error 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: bad goto syntax

Compare your use of *goto* with an example. This message says that you did not specify where you wanted to *goto* with a label:

```
    goto label;  
    ...  
label:  
    ...
```

It is not possible to `goto` just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in *do-while*

The body of a *do-while* may consist of one statement or several statements enclosed in braces. A *while* conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, don't forget the *while* conditional.

72: 'for' syntax error: missing first semicolon

This error focuses on another control flow statement, the *for*. The keyword, *for*, must be followed by parentheses. In the parentheses belong three expressions, any or all of which may be null. For the sake of clarity, C requires that the two semicolons which separate the expressions be retained, even if all three expressions are empty.

```
    for ( ;                               /* an infinite loop which does */  
        ;                               /* absolutely nothing */
```

Error 72 signifies that the compiler didn't find the first semicolon within the parentheses.

73: 'for' syntax error: missing second semicolon

This error is similar to error 72; it means that the compiler didn't find the second semicolon within the parenthesized expression following the 'for'.

74: case value must be integer constant

Strictly speaking, each value in a *case* statement must be a constant of one of three types: *char*, *int* or *unsigned*. This is similar to the rule for a *switched* variable. In the following example, a float must be cast to an *int* in order to be *switched*; however, notice that the programmer did not check his case statements. The second case value is invalid, and the code will not compile.

```
float k = 5.0;
switch((int)k) {
case 4:
    printf("good case value\n");
    break;
case 5.0:
    printf("bad case value\n");
    break;
}
```

The programmer must replace "case 5.0:" with "case 5".

75: missing colon on case

This should be straightforward. If the compiler accepts a case value, a colon should follow it. A semi-colon must not be accidentally entered in its place.

76: too many cases in switch

The compiler reserves a limited number of spaces in an internal table for *case* statements. If a program requires more cases than the table initially allows, it becomes necessary to tell the compiler what the table value should be changed to. It is not necessary to know exactly how many are needed; an approximation is sufficient, depending on the requirements of the situation.

77: case outside of switch

The keyword, *case*, belongs to just one syntactic structure, the *switch*. If "case" appears outside the braces which contain a switch statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon

This message indicates that a colon is missing after the keyword, *default*. Compare error 75.

79: duplicate default

The compiler has found more than one *default* in a *switch*. Switch will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a case statement. It is analogous to the *else* companion to the conditional, *if*. Just as there is one *else* for every *if*, only one default case is allowed in a switch statement. However, unlike the *else* statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, *default*, is used just like *case*. It must appear within the brackets which delimit the switch statement.

81: break/continue error

Break and continue are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a switch statement. But when the keywords, *break* or *continue*, are used outside of these contexts, this message results.

82: illegal character

Some characters simply do not make sense in a C program, such as '\$' and '@'. Others, for instance the pound sign (#), may be valid only in particular contexts.

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, file D is not allowed to have a #include in the compilation of file A.

file A	file B	file C	file D
#include "B"	#include "C"	#include "D"	

84: too many array dimensions

An array is declared with too many dimensions. This error should appear in conjunction with error 11.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an extern declaration and an array initialization. The value of any dimension which is not the left-most must be given.

extern char array[][12];	/* correct */
extern char badarray[5][];	/* wrong */

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as 'a' or 'ab'. There is no analog to a null string, so "" (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (\b, \n, \t etc.) are singular,

so that the following are valid: '\n', '\na', 'a\n'; 'aaa' is invalid.

88: not a structure

Occurs only under compilation without the -S option. A name used as a structure does not refer to a structure, but to some other data type.

```
int i;
i.member = 3;          /* error 88 */
```

89: invalid storage class

A globally defined variable cannot be specified as register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: illegal use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: illegal type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;
float j;
char *ptr;
...
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type *char* and *short* become *int*, and *float* becomes *double*. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a *float* will evaluate to a *double*.

hierarchy of types:

```
double <-- float
long
unsigned
int <-- short, char
```

This error can also be caused by an attempt to return a structure, since the structure is being cast to the type of the function, as in:

```
int func()
{
    struct tag sam;
    return sam;
}
```

93: illegal expression type for switch

Only a *char*, *int* or *unsigned* variable can be switched. See the example for error 74.

94: bad argument to define

An illegal name was used for an argument in the definition of a macro. For a description of legal names, see error 65.

95: no argument list

When a macro is defined with arguments, any invocation of that macro is expected to have arguments of corresponding form. This error code is generated when no parenthesized argument list was found in a macro reference.

```
#define getchar() getc(stdin)
...
c = getchar;                      /* error 95 */
```

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

```
#define reverse(x,y,z) (z,y,x)
func(reverse(i,,k));
```

97: obsolete [see error 19]**98: not enough args in macro reference**

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define exchange(x,y) (y,x)
func(exchange(i));                /* error 98 */
```

99: internal [see error 4]**100: internal** [see error 4]**101: missing close parenthesis on macro reference**

A right (closing) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between #if and #else is analogous to that which exists between the control flow statements, if and else. Obviously, much depends upon the relative placement of the statements in the code. However, #if blocks must always be terminated by #endif, and the #else statement must be included in the block of the #if with which it is associated. For example:

```
#if ERROR > 0
    printf("there was an error\n");
#else
    printf("no error this time\n");
#endif
```

#if statements can be nested, as below. The range of each #if is determined by a #endif. This also excludes #else from #if blocks to which it does not belong:

```
#ifdef JAN1
    printf("happy new year!\n");
#if sick
    printf("i think i'll go home now\n");
#else
    printf("i think i'll have another\n");
#endif
#else
    printf("i wonder what day it is\n");
#endif
```

If the first #endif was missing, error 103 would result. And without the second #endif, the compiler would generate error 107.

104: #endif with no #if

#endif is paired with the nearest #if, #ifdef or #ifndef which precedes it. (See error 103.)

105: #endasm with no #asm

#endasm must appear after an associated #asm. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a #endasm without having found a previous #asm. If the #asm was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the `#asm` keyword must not appear between a paired `#asm/#endasm`. When a piece of in-line assembly is augmented for temporary purposes, the old `#asm` and `#endasm` can be enclosed in comments as place-holders.

```
#asm
/* temporary asm code */
/* #asm      old beginning */
/* more asm code */
#endasm
```

107: missing #endif

A `#endif` is required for every `#if`, `#ifdef` and `#ifndef`, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first `#endif`. Backtrack to the previous `#if` and form the pair. Assign the next `#endif` with the nearest unpaired `#if`. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a `#endasm` in all cases. `#asm` must always be paired with a `#endasm`.

109: #if value must be integer constant

`#if` requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers,

```
#if DIFF >= 'A'-'a'
#if (WORD &= ~MASK) >> 8
#if MAR | APR | MAY
```

are all legal expressions for use with `#if`.

110: invalid use of colon operator

The colon operator occurs in two places: 1. following a question mark as part of a conditional, as in `(flag ? 1 : 0)`; 2. following a label inserted by the programmer or following one of the reserved labels, *case* and *default*.

111: illegal use of a void expression

This error can be caused by assigning a *void* expression to a variable, as in this example:

```
void func();
int h;

h = func(arg);
```

112: illegal use of function pointer

For example,

```
int (*funcptr) ();
...
funcptr++;
```

funcptr is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not subject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

This simply means that, in a *switch* statement, there are two *case* values which are the same. Either the two *cases* must be combined into one, or one of them must be discarded. For instance:

```
switch (c) {
case NOOP:
    return (0);
case MULT:
    return (x * y);
case DIV:
    return (x / y);
case ADD:
    return (x + y);
case NOOP:
default:
    return;
}
```

The case of NOOP is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define islow(n) (n>=0&& n<5)
...
#define islow(n) (n>=0&& n<=5)
```

The macro, *islow*, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define islow(n) n>=0&& n<=5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define NULL 0
```

```
...
```

```
#define NULL 0
```

But these are different from:

```
#define NULL ' '
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define int foo
```

If you have a variable which may be either, for instance, a short or a long integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef LONGINT
    long i;
#else
    short i;
#endif
```

Another possibility is through a *typedef*:

```
#ifdef LONGINT
    typedef long VARTYPE;
#else
    typedef short VARTYPE;
#endif

VARTYPE i;
```

116: field width must be > 0

A field in a bit field structure can't have a negative number of bits.

117: invalid 0 length field

A field in a bit field structure can't have zero bits.

118: field is too wide

A field in a bit field structure can't have more than 16 bits.

119: field not allowed here

A bit field definition can only be contained in a structure.

120: invalid type for field

The type of a bit field can only be of type *int* or *unsigned int*.

121: ptr/int conversion

The compiler issues this warning message if it must implicitly convert the type of an expression from pointer to *int* or *long*, or vice versa.

If the program explicitly casts a pointer to an *int* this message won't be issued. However, in this case, error 122 may occur.

For example, the following will generate warning 121:

```
char *cp;
int i;

...
i = cp; /* implicit conversion of char * to int */
```

When the compiler issues warning 121, it will generate correct code if the sizes of the two items are the same.

122: ptr & int not same size

If a program explicitly casts a pointer to an *int*, and the sizes of the two items differ, the compiler will issue this warning message. The code that's generated when the converted pointer is used in an expression will use only as much of the least significant part of the pointer as will fit in an *int*.

123: function ptr & ptr not same size

If a program explicitly casts a pointer to a data item to be a pointer to a function, or vice versa, and the sizes of the two pointers differ, the compiler issues this warning message.

If the program doesn't explicitly request the conversion, warning 124 will be issued instead of warning 123.

124: invalid ptr/ptr assignment

If a program attempts to assign one pointer to another without explicitly casting the two pointers to be of the same type, and the types of the two pointers are in fact different, the compiler will issue this warning message.

The compiler will generate code for the assignment, and if the sizes of the two pointers are the same, the code will be correct. But if the

sizes differ, the code may not be correct.

125: too many subscripts or indirection on integer

This warning message is issued if a program attempts to use an integer as a pointer; that is, as the operand of a star operator.

If the sizes of a pointer and an *int* are the same, the generated code will access the correct memory location, but if they don't, it won't.

For example,

```
char c;  
long g;  
*0x5c=0; /* warning 125, because 0x5c is an int */  
c[i]=0; /* warning 125, because c+i is an int */  
g[i]=0; /* error 12, because g+i is a long */
```


3. Fatal Compiler Error Messages

If the compiler encounters a "fatal" error, one which makes further operation impossible, it will send a message to the screen and end the compilation immediately.

Out of disk space!

There is no room on the disk for the output file of the compiler. Previous disk files will not be overwritten by the compiler's assembly language output. To make room on the disk, it is usually sufficient to remove unneeded files from the disk.

unknown option:

The compiler has been invoked with an option letter which it does not recognize. The manual explicitly states which options the compiler will accept. The compiler will specify the invalid option letter.

duplicate output file

If an output file name has been specified with the -o option and that file already exists on the disk, the compiler will not overwrite it. -O must specify a new file.

too few arguments for -o option

The compiler expected to find the output filename following the "-o", but didn't find it. The output file name must follow the option letter and the name of the file to be compiled must occur last in the command line.

Open failure on input

The input file specified in the command line does not exist on the disk or cannot be opened. A path or drive specification can be included with a filename according to the operating system in use.

No input!

While the compiler was able to open the input file given in the command line, that file was found to be empty.

Open failure on output

The compiler was unable to create an output file. On some systems, this error could occur if a disk's directory is full.

Local table full! (use -L)

The compiler maintains an internal table of the local variables in the source code. If the number of local symbols in use exceeds the available entries in the table at any time during compilation, the compiler will print this message and quit. The default size of the local symbol table (40 entries) can be changed with the -L option for the

compiler. Local variables are those defined within braces, i.e., in a function body or in a compound statement. The scope of a local variable is the body in which it is defined, that is, it is defined until the next right brace at its own nesting level.

Out of memory!

Since the compiler must maintain various tables in memory as well as manipulate source code, it may run out of memory during operation. The more immediate solution is to vary the sizes of the internal tables using the appropriate compiler options. Often, a compilation will require fewer than the default number of entries in a particular table. By reducing the size of that table, memory space is freed up during compile time. The amount of memory used while compiling does not affect the size or content of the assembly or object file output. If this strategy fails to squeeze the compilation into the available memory, the only solution is to divide the source file into modules which can be compiled separately. These modules can then be linked together to produce a single executable file.

Aztec CG65 Cross Development System
Host: PCDOS/MSDOS Target: 65xx-based systems
Version 3.2
Release Document
25 Aug 1986

This package contains the PCDOS/MSDOS-to-65xx cross development version of Aztec CG65, v3.2. This release document is divided into the following sections:

1. Product Overview
2. Packaging
3. Known Bugs
4. Technical Support

If changes have been made to Aztec CG65 since this release document was printed, information about the changes will be in a file named *read.me* on the first distribution disk.

1. Product Overview

Aztec CG65 is a set of programs that translate C language programs into code that can be executed by a 65xx microprocessor. Using the Aztec *hex65* utility, the generated code can optionally be burned into ROM.

With this version of Aztec CG65, you develop programs on a PCDOS or MSDOS system.

The package contains the following:

- Disks in IBM PCDOS format, containing the Aztec CG65 software;
- A manual that describes the Aztec CG65 software;
- The commercial version of Aztec C86 for PCDOS, with which you can do native development on an IBM PC (i.e. develop and execute programs on an IBM PC).

2. Product Packaging

2.1 Executable programs

CG65.EXE	6502/65C02 compiler
AS65.EXE	6502/65C02 assembler
CC1.EXE	Pseudo code compiler
AS1.EXE	Pseudo code assembler
LN65.EXE	Linker
HEX65.EXE	Intel hex code generator

OPTINT65.EXE	Pseudo code optimizer
LB65.EXE	Object module librarian
CNM65.EXE	Object module summarizer
OBD65.EXE	Object module displayer
SQZ65.EXE	Object module compressor
ORD65.EXE	Object module library utility
ARCV.COM	Dearchiver
MKARCV.COM	Archiver
MAKE.EXE	Program maintenance utility
HD.EXE	File dumper
CRC.EXE	File CRC generator

Note: the actual names of the object module utilities differ slightly from their documented names; i.e. from the names by which the manual refers to them. The actual name of an object module utility is derived from its documented name by appending "65". Thus, the documented names of the object module utilities are *lb*, *cnm*, *obd*, *sqz*, and *ord*, while their actual names are *lb65*, *cnm65*, *obd65*, *sqz65*, and *ord65*.

2.2 Header files

Several 'header files' are provided, which have extension *.h*, and which a C source program accesses using the *#include* statement.

2.3 Source Archives

Several files containing source archives are provided. Some are used to make the object module libraries; others, which are not absolutely necessary for the development of C programs, are provided because you may find them useful. The program *arcv* unpacks an archive's contents into separate files. For a description of *arcv*, see the *Utility Programs* chapter.

The archives that are used to generate libraries are:

DEV.ARC	Device driver functions
FLT.ARC	Floating point functions
LIBMAKE.ARC	Files used to generate libraries
MCH65.ARC	Low-level, 65xx-specific functions
MISC.ARC	Miscellaneous C-language functions
OVLY.ARC	Overlay functions
PRODOS.ARC	Apple // ProDOS functions
ROM.ARC	ROM support functions, and <i>hex65</i> source
STDIO.ARC	Standard I/O functions
TIME.ARC	Time functions

The other source archives are:

CONFIG.ARC	Device configuration program, <i>config</i>
TTY.ARC	Terminal emulator that runs on an Apple //
XFER.ARC	File transfer program, <i>xfer</i>

Documentation for *config*, *tty*, and *xfer* are in their source archives.

2.4 Sample programs

The file *exmpl.c* contains a sample C program.

3. Known bugs

3.1 *sqz65*

sqz65, the program that compresses an object module, should only be used on modules that are going to be put in a library.

3.2 *optint65*

Don't use *optint65* on *cci*-compiled modules that contain *float* variables - for such programs it generates incorrect code. You can use it on modules that contain *double* variables.

The-library routines don't contain *float* variables, so *optint65* can be used to generate the *cci*-compiled libraries *ci.lib* and *mi.lib*.

4. Technical support information

While we do our best to ship problem free software, sometimes the unknown does happen and problems occur. Manx has a technical support staff ready to help you out if you should encounter problems while using our software. At the very end of this document is a discussion of how to make the most out of the technical support that Manx offers. In addition, we have added problem report forms for the reporting of any problems you may encounter with our software.

Using MANX Technical Support

We have put together a set of guidelines to help you take the most advantage of the technical support service offered by MANX. We ask that you read and follow these guidelines to enable us to continue to give you quality technical support.

Have everything with you.

Try to be organized. When using our phone support, have everything you need with you at the time you call. Our goal is to get you the help you need without keeping you on the phone too long. This can save you a lot of time, and if we can keep the calls as short as possible we can take more calls in the day. This can be to your advantage on days when we are busy and it's hard to get through. Also, *have the following information ready* when you call technical support. We will ask you for this information first.

- * *Your name.* This is necessary in case we need to get back to you with additional information.
- * *Phone number.* In case we have additional information we will be able to contact you. This will never be given to anyone, so you need not worry.
- * *The product* you are using, and the *serial number.* If you have a cross compiler please tell us both host and target, even if the problem is with just one side of the system.
- * *The revision of the product* you are using. This should include a letter after the number: i.e. 3.20d or 1.06d. **THIS IS VERY IMPORTANT.** The full version number may be found on your distribution disks or when you run the COMPILER.
- * *The operating system* you are using, and also the version.
- * *The type of machine* you are using.
- * Anything interesting about your machine configuration. i.e. ram disk, hard disk, disk cache software etc.

Know what questions you wish to ask.

If you call with a usage question please try to have your questions narrowed down as much as possible. It is easier and quicker for all to answer a specific question than general ones.

Isolate the code that caused the problem.

If you think you have found a bug in our software, try and create a small program that reproduces the problem. If this program is small enough we will take it over the phone, otherwise we would prefer that you mail it to us, using the supplied problem report, or leave it on one of our bbs systems. Once we receive a "bug report" we will attempt to reproduce the problem and if successful we will try to have it fixed in the next release. If we can not reproduce the problem we will contact you for more information.

Use your C language book and technical manuals first.

We have no qualms about helping you with your general C programming questions, but please check with a C language programming book first. This may answer your question quicker and more thoroughly. Also, if you have questions about machine specific code, i.e. interrupts or dos calls, check with that machine's technical reference manual and/or operating system manual.

When to expect an answer.

A normal turn around time for a question is anywhere from 2 minutes to 24 hours, depending on the nature of the question. A few questions like tracing compiler bugs may take a little longer. If you can call us back the next day, or when the person you talk to in technical support recommends, we will have an in-depth answer for you. But normally we can answer your questions immediately.

Utilize our mail-in service.

It is always easier for us to answer your question if you mail us a letter (We have included copies of our problem report form for your use). This is especially true if you've found a bug with our compiler or other software in our package. If you do mail your question in, try to include all of the above information, and/or a disk with the problem. Again, please write small test programs to reproduce possible bugs. The address for mail-in reports is P.O. Box 55, Shrewsbury, N.J. 07701. If you have questions/problems concerning C Prime or Apprentice C, mail them to P.O. Box 8, Shrewsbury, N.J. 07701.

Updates, Availability, Prices.

If you have any questions about updates, availability of software, or prices, please call our order desk. They can help you better and faster. You can reach them at...

Outside N.J. --> 1-800-221-0440
Inside N.J. --> 1-201-542-2121 (also for outside the U.S.A.)

Bulletin board system.

For users of Aztec C we have a bulletin board system available. The number is ...

1-(201)-542-2793 This is at 300/1200 bps. (all products)

Answer the questions that will be asked after you are connected. When this is done you will be on the system with limited access. To gain a higher access level send mail to SYSOP. Include in this information your serial number and what product you have. Within approximately 24 hours you should have a higher access level, provided the serial number is valid. This will allow you to look at the various information files and upload/download files.

To use the bulletin board best, please do not put large (> 8 lines) source files onto the news system, which we use for an open forum question/answer area. Instead, upload the files to the appropriate area, and post a news item explaining the problem you are having. Also, the smaller the test program, the quicker and easier it is for us to look into the problem, not to mention the savings of phone time.

When you do post a news item, please date it and sign it. This will be very helpful in keeping track of questions. Try to do the same with uploaded source files.

Phone support, number and hours.

Technical support for Aztec C is available between 9:00 am and 6:00 pm eastern standard time at 1-(201)-542-1795. Phone support is available to registered users of Aztec C with the exception of the Apprentice C and C Prime products. For those products, please use the mail-in support service and send questions/problems to P.O. Box 8, Shrewsbury, N.J. 07701.

These guidelines will aid us in helping you quickly through any roadblocks you may find in your development. Thanks for your cooperation.

MANX Problem Report

Date: ____/____/____

Name: _____

Phone #: 1-(____)-____-____

Company : _____

Address : _____

Product : c86-PC____ c86-CPM86____ c68k____
c68k-Am____ cII____ c80____
c65-ProDos____ c65-Dos3.3____
cross: _____

VERSION #: _____ Serial #: _____

Op. - sys.: _____ Machine Config.: _____

Send this form to :

Manx Software Systems	(C Prime/Apprentice C only):
P.O. Box 55	MANX Software Systems
Shrewsbury, N.J. 07701	P.O. Box 8
	Shrewsbury, N.J. 07701

or call tech support at 1-201-542-1795 between 9am - 6pm EST.
(Sorry, phone support not available for the C Prime/Apprentice C product.)

Description of problem --
(include what has already been attempted to fix it)
(use the reverse side of this sheet if needed.)

MANX Problem Report

Date: ____/____/____

Name: _____

Phone #: 1-(____)-____-____

Company : _____

Address : _____

Product : c86-PC____ c86-CPM86____ c68k____
c68k-Am____ cII____ c80____
c65-ProDos____ c65-Dos3.3____
cross: _____

VERSION #: _____ Serial #: _____

Op. - sys.: _____ Machine Config.: _____

Send this form to :

Manx Software Systems
P.O. Box 55
Shrewsbury, N.J. 07701

(C Prime/Apprentice C only):
MANX Software Systems
P.O. Box 8
Shrewsbury, N.J. 07701

or call tech support at 1-201-542-1795 between 9am - 6pm EST.
(Sorry, phone support not available for the C Prime/Apprentice C
product.)

Description of problem --
(include what has already been attempted to fix it)
(use the reverse side of this sheet if needed.)

MANX

software systems

End User License Agreement

September 1982

Use and possession of this software package is governed by the following terms:

DEFINITIONS — these definitions shall govern:

- A. Supplier means MANX SOFTWARE SYSTEMS, P.O. Box 55, Shrewsbury, NJ 07701, the author and owner of all rights to this SOFTWARE.
- B. "Customer" means the individual purchaser, its agents and employees, and the company CUSTOMER works for and its agents and employees, if the company paid for this software.
- C. "Computer" is the single computer on which customer uses this program. Multiple CPU systems requires supplementary licenses.
- D. "Software" is the set of computer programs in this package, regardless of the form in which Customer may subsequently use it, which Customer may make to it.
- E. "License" means this agreement and the rights and obligations which it creates under the United States Copyright Law and New Jersey laws.
- F. "Runtime Library" is the set of copyrighted Manx Software Systems language subroutines, provided with each language compiler, a portion of which must be linked to and become part of a Customer program for that program to run on the Computer.

The Supplier grants Customer the right to use this serialized copy of the Software so long as customer complies with the terms of the license. Read the licensing agreement carefully. If you do not agree to the terms contained in this license, return the diskette package UNOPENED to the Seller from whom you purchased it, who will refund your money subject to the conditions of this Agreement.

If the unopened diskette package is not returned to the Seller within fourteen (14) days of delivery the Customer will be deemed to have accepted all terms of the license agreement and will be bound thereby. Return of a diskette package that has been opened, damaged or otherwise tampered with shall also operate as an acceptance of the license terms by the Customer and no money will be refunded. Seller may also deduct the price of the manual, shipping and handling expenses, and other such expenses that may be incurred in processing returned Software.

When you open the package, you need to sign and return the Registration Card in order to become a registered user, and thereafter to receive a number of substantial benefits, including support and notice of updated materials. Supplier does not support unregistered users.

CONDITIONS OF LICENSE

The Supplier agrees to grant, and the Customer agrees to accept, on the following terms and conditions, a non-exclusive license to use the software programs herein delivered with this agreement.

All updates of MANX SOFTWARE received by Customer from Supplier, or from any other source, are subject to the terms and conditions of this license agreement and Customer hereby agrees to be bound thereby.

DURATION:

This agreement is effective from the date of receipt of the Software and shall remain in force unless terminated by the Supplier or by the Customer as provided below.

LICENSE:

Each program license granted under this Agreement authorizes the Customer to use the Licensed Software in any machine readable form on the Computer designated in the agreement. A separate license is required for each Computer on which the licensed Software will be used.

The Customer has no right to transfer, assign, or sublicense the licenses, Software, materials, or this Agreement without prior written consent from the Supplier. No right to print, copy, reproduce or in any other manner duplicate, in whole or in part, the Licensed Software or documentation is granted to Customer except as is hereinafter expressly provided. Additional copies of printed materials may be acquired from the Supplier.

Customer understands that unauthorized reproduction of copies of the Software and/or unauthorized transfer of any copy may be a serious crime, as well as subjecting Customer to damages and attorney fees. Customer may not transfer any copy of the Software to another person unless Customer transfers all copies, including the original, and advises Supplier of the name and address of that person, who must sign a copy of the registration card card, pay the then current transfer fee, and agree to the terms of this License in order to use the Software. Supplier will provide additional copies of the card and License upon request. Supplier has the right to terminate the License, to trace serial numbers, and to take legal action if these conditions are violated. Supplier has the right deny permission to transfer the Software.

The Customer agrees not to provide or disclose the Licensed Software including, but not limited to, program listings, object code, and source code, in any form, to any person other than Customer, the Supplier or the

Suppliers agents and employees except for the purposes specifically related to the Customer's use of the Licensed Software on the licensed computer.

Under no circumstances shall the Customer provide public access to the Licensed Software in whole or in part, transformed or untransformed; including, but not limited to, computer time sharing networks, periodicals, newspapers, or any other accessible or distributed media.

The Customer agrees to take all reasonable steps to insure that the license terms and conditions will be made known to anyone who uses in whole or in part the Licensed Software.

The Customer agrees to insure that all materials that could lead to the use of the licensed Software in a manner that violates this Agreement will be erased or destroyed when they are no longer needed.

PERMISSION TO COPY OR MODIFY LICENSED SOFTWARE

The Customer is permitted to copy and modify the programs licensed hereunder provided such modification is required for use of the Licensed Software in the Customer's environment on the licensed computer. All modifications or copies of the Licensed Software, regardless of how or by whom they were made shall be the property of the Supplier. Supplier's proprietary interest shall not include the media on which the changes to the Licensed Software are recorded. All copies or modifications of the Licensed Software are restricted to the licensed Computer and are bound by the same terms and conditions of this Agreement as the original Licensed Software delivered hereunder.

The Customer agrees to reproduce and include the copyright notices on all copies, in whole or in part, in any form, including partial copies in modifications of Licensed Software hereunder.

The Customer agrees to record and retain records of any and all copies or modifications made to the Licensed Software until they are destroyed. The Customer agrees to provide these records to the Supplier within 30 days of written request for same.

As an exception to the preceding, Customer is granted the right to include portions of the Supplier's Runtime Library in Customer developed programs, called Composite Programs, and to use, distribute and license such Composite Programs. As an express condition to the use of the Runtime Library, customer agrees to indemnify and hold Supplier harmless from all claims by Customer and third parties arising out of the use of Composite Programs.

PERMISSION TO USE LICENSED SOFTWARE ON ALTERNATE COMPUTER(S)

Use of the Licensed Software is by the express terms of this Agreement restricted to the single computer designated in the registration card. The customer may obtain separate license(s) to use the Licensed Software on additional Computers under the control and operation of Customer by completing a registration card for each computer and returning same to Supplier with the prevailing multiple license fee. The terms of this License Agreement will be thereby transferred to the newly registered Computer (s) and the parties shall be bound thereby.

DISCONTINUANCE:

Within 30 days of the date of discontinuance of any license under this Agreement, the Customer will furnish the Supplier with a certificate certifying that all of the Licensed Programs, including modifications, copies, the original supplied with this Agreement, and any and all derivatives have been destroyed.

DISCLAIMER OF WARRANTY AND LIMITATION OF LIABILITY:

The Supplier makes no warranties with respect to the Licensed Program(s), EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL THE SUPPLIER BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF THE SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT WILL SUPPLIER LIABILITY EXCEED THE ORIGINAL PURCHASE PRICE OF THE LICENSED PROGRAM.

GENERAL:

If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, they are to that extent deemed omitted and the balance of this agreement shall remain in full force and effect.

This is the complete and exclusive agreement between the Supplier and Customer and supersedes all proposals, oral or written, and all other communications between the parties relating to the subject matter of the Agreement. This Agreement may not be modified orally.

This Agreement will be governed by the laws of the State of New Jersey.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

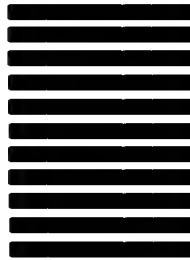
FIRST CLASS PERMIT NO. 67 RED BANK, N.J.

POSTAGE WILL BE PAID BY

MANX[®]

software systems

Box 55, Shrewsbury, N.J. 07701



Registration Card

C 2089

I have read the Software Licensing Agreement for the AZTEC C Compiler and the other Licensed Programs supplied with the Agreement and agree to abide by the terms contained in it:

Company _____ Date _____

Name _____ Phone No. _____

Address _____

City _____ State _____ Zip _____

Country _____ Product *C26C* Version *2.20C*

Please fill out the registration information before opening the diskette package. Upon receipt of this registration by MANX Software Systems you will become a registered AZTEC C user.

Signature _____

☐ Initial Registration

INDEX

Order of chapters in manual

System Dependent Chapters

<i>title</i>	<i>code</i>
Overview	ov
Tutorial Introduction	tut
The Compilers	cc
The Assemblers	as
The Linker	ln
Utility Programs	util
Library Generation	libgen
Technical Information	tech

System Independent Chapters

Overview of Library Functions	libov
System-Independent Functions	lib
Style	style
Compiler Error Messages	err

Index

Index	index
-------------	-------

6502 stack tech.14,15
 65xx link options ln.9,15
 65xx compiler options
 cc.7,11
 __main function libgen.3,4

A

absolute value lib.16
 accessing devices libov.8
 acos lib.59-60
 adding modules to a library
 util.18-20
 agetc libgen.3,5;lib.25-26
 aputc libgen.3,5;lib.41-42
 arcv & mkarcv - source
 dearchiver & archiver util.4
 arguments as.6
 arithmetic operators as.6
 array subscripting style. 18
 asin lib.59-60
 assembler operating
 instructions as.3
 assembler options as.5
 -c as.5
 -i as.4,5,9
 -l as.4,5
 -o as.4,5
 -zap as.5
 assign buffer to a stream
 lib.56
 atan lib.59-60
 atan2 lib.59-60
 atof lib.8
 atoi lib.8
 atol lib.8
 automatic variables cc.7,13

B

base address ln.9,12-14
 boolean expressions
 style. 16-17
 buffered binary input
 lib.20-21
 buffered output lib.20-21
 buffering libov.10-11
 build and unbuild real numbers

lib.22
 building the libraries
 libgen.5

C

c source file cc.3-5,20
 calloc lib.31-32
 case table cc.7,10,11
 cbreak libov.21
 ceil lib.16
 ceiling lib.16
 change current position
 within a file lib.29-30
 char cc.17
 character classification
 funtions lib.11
 character-oriented input
 libov.18
 clearerr lib.15
 close lib.9
 close a device or a file
 lib.9
 closing streams
 lib.14;libov.9
 cnm - display object
 file info util.5-8
 code area tech.4
 code section tech.20,27
 command line arguments
 libov.4-6
 comments style. 17
 common problems style.15-19
 compiler error checking
 cc.23
 compiler operating instructions
 instructions cc.3
 compiler options cc.3,7
 +b cc.7,12,13
 +c cc.7,12
 +g cc.7,12;libgen.6
 +l cc.7,13
 -a cc.5,7
 -b cc.7,23
 -d cc.7,8;tech.8,13
 -e cc.7,10
 -i cc.6-8
 -l cc.7,9,10

- o cc.4,5,7
- s cc.7,8
- t cc.5,7
- y cc.7,10,11
- z cc.7,11
- console i/o libov.17-21
- constants as.7
- convert ascii to numbers lib.8
- convert floating point to
 ascii lib.8
- cos lib.59-60
- cosh lib.61
- cotan lib.59-60
- crc - utility for generating
 the crc for files util.9
- creat lib.10
- create a new file lib.10
- creating an object code
 file cc.4

D

- data formats cc.17
- default mode libov.7,17,20
- defensive programming
 style.10
- deleting modules util.19
- device i/o libov.7
- device i/o utilities
 lib.28
- directives as.5-9
 - bss as.8;util.7
 - cseg as.7
 - dseg as.7
 - end as.7
 - entry as.8,9;ln.12,13
 - equ as.6,8;util.6
 - fcbl as.9
 - fcc as.9
 - fdb as.9
 - global as.8;ln.11,13;
 tech.20;util.8
 - instxt as.4,5,9
 - public as.8;ln.11,13;
 tech.14,20;util.6
 - rmb as.9
- directories as.4,5,9

- double cc.17
- dynamic buffer allocation
 libov.11,22

E

- echo mode libov.21
- error messages from ovloader
 tech.12
- error processing libov.23-24
- exp lib.12-13
- exponential lib.12-13
- expression table cc.7,10
- extracting modules
 from a library util.23

F

- fabs lib.16
- fclose lib.14
- fdopen lib.17-19
- feof lib.15
- ferror lib.15
- fflush lib.14
- fgets lib.27
- file i/o libov.6,9-13,15
- fileno lib.15
- float cc.17-19
- floating point exceptions
 cc.18
- floor lib.16
- filter cc.18,19
- flush a stream lib.15
- fopen lib.17-19
- format lib.37-40
- formatted input conversion
 lib.49-55
- formatted output conversion
 functions lib.37-40
- fprintf lib.37-40
- fputs lib.43
- fread lib.20-21
- free lib.31-32
- freopen lib.17-19
- frexp lib.22
- fscanf lib.49-55
- fseek lib.23-24
- ftell lib.23-24

ftoa lib.8
 functions calls style.13-14
 fwrite lib.20-21

G

get a string from
 a stream lib.27
 getc lib.25-26
 getchar lib.25-26
 gets lib.27
 getw lib.25-26
 global variables cc.16,17

H

hd - hex dump utility
 util.10
 header section tech.18
 heap tech.5,6,12
 help util.14,24
 hex65 - intel hex generator
 util.11-13;libgen.4;tech.5
 hyperbolic functions lib.61

I

in-line assembly language
 code cc.20
 incl65 environment variable
 as.5,9;cc.6
 index lib.62-63
 initialized data area
 tech.4,5
 inquiries lib.15
 ioctl lib.28
 isalnum lib.11
 isalpha lib.11
 isascii lib.11
 isatty lib.28
 isctrl lib.11
 isdigit lib.11
 islower lib.11
 isprint lib.11
 ispunct lib.11
 isspace lib.11
 isupper lib.11

L

labels as.6
 lb - object file librarian
 util.14-24;libgen.5-7
 ldexp lib.22
 learning c idioms style. 3
 libraries ln.4-8,10
 line continuation cc.14
 line-oriented input
 libov.17-18
 linker options ln.9
 +c ln.9,15;
 tech.5,9,10,12
 +d ln.9,15;
 tech.5,9,10,12
 +h ln.9,15
 -b ln.9,13,14
 -c ln.9,12-14
 -d ln.9,12-14
 -f ln.9-11
 -l ln.8-10
 -m ln.9,11
 -n ln.9,12
 -o ln.7-10,14
 -r ln.9,14,15;tech.9-11
 -t ln.9,11
 -u ln.9,12
 -v ln.9,12
 linking process ln.4
 literal table cc.7,11
 local symbol table cc.9-10
 log lib.12-13
 logarithm lib.12-13
 long cc.17
 longjmp lib.57-58
 loader items tech.21-27
 lseek lib.29-30

M

machine-independent
 options cc.7,8
 macro/global
 symbol table cc.11
 macros util.31-34,39,40
 make - program maintenance
 utility util.25-42;libgen.3,5-7
 makefile util.25-42;libgen.3-5

makefile syntax util.36
malloc lib.31-32
memory allocation lib.31-32
memory organization tech.4
missing semicolon style. 15
modf lib.22
modularity style. 7
moving modules within
 a library util.18
movmem lib.33
mpu... symbols cc.22

N

nesting errors style. 17
nodelay libov.17
non-local gotto lib.57-58

O

object library format
 tech.27
opcodes as.6
open a stream lib.17-19
open lib.34-36
opening files and devices
 libov.2,6,9
operating instructions as.3
optint65 - pseudo-code optimizer
 util.44
options for segment
 address specification ln.9,12
order in a library util.17
order of evaluation style. 16
order of library modules ln.5
overlays tech.5,7-12
ovloader tech.8,10-13

P

passing data to functions
 style. 18
pointer cc.14,15,17,19
pow lib.12-13
power lib.12-13
pre-opened devices libov.4
printf lib.37-40

program organization
 tech.6
pseudo stack
 libgen.4;tech.4,5,14,15
push a character back
 into input stream lib.65
put a character string
 to a stream lib.43
putc lib.41-42
putchar lib.41-42
puterr lib.41-42
puts lib.43
putw lib.41-42

Q

qsort lib.44-45

R

ran lib.46
random i/o libov.6,10
random number generator
 lib.46
raw mode libov.20-21
read lib.47
readable code style. 5
realloc lib.31-32
rebuilding a library
 util.23
register variables
 cc.12,20
rename a disk file
 lib.48
replacing modules util.20
reposition a stream
 lib.23-24
reserved words cc.16
rewriting the functions
 libgen.3
rindex lib.62-63
rules
 util.25,28-30,32-34,36,39,40
run-time errors style. 12

S

scanf lib.49-55
 sequential i/o libov.6,10
 setbuf lib.56
 setjmp lib.57-58
 setmem lib.33
 sgtty fields libov.19
 shared data style. 19
 sin lib.59-60
 sinh lib.61
 sort an array lib.44-45
 special symbols cc.15
 sprintf lib.37-40
 sqrt lib.12-13
 square root lib.12-13
 sqz - squeeze an object
 library util.46
 sscanf lib.49-55
 standard i/o libov.9-13
 standard i/o functions
 libgen.4,5;libov.12-13
 start-up function libgen.3
 startup routine ln.12,13
 strcat lib.62-63
 strcmp lib.62-63
 strcpy lib.62-63
 stream status lib.15
 string merging cc.15
 string operations lib.62-63
 string table cc.11
 strlen lib.62-63
 strncat lib.62-63
 strncmp lib.62-63
 strncpy lib.62-63
 structure assignment cc.14
 structured programming
 style. 7
 supported language features
 cc.14
 swapmem lib.33
 symbol names cc.16
 symbol table ln.9,11,14,15
 symbol tables tech.19
 syntax as.5,6
 system-independent programs
 libov.18

T

tan lib.59-60
 tanh lib.61
 tolower lib.64
 top-down programming
 style. 8-9
 toupper lib.64
 trigonometric functions:
 lib.59-60

U

unbuffered i/o libov.14-16
 unbuffered i/o functions
 libgen.3-5
 unbuffered and standard
 i/o calls libov.7
 ungetc lib.65
 uninitialized data area
 libgen.4;tech.5
 uninitialized variables
 style.15
 unlink lib.66
 using the linker ln.7
 utility programs util.3

V

void data type cc.14

W

write lib.67

Z

zero page usage
 cc.12;libgen.6

